

Technische Universität Ilmenau
Department of Computer Science and Automation
Institute for Applied Computer Science and Media Communications
Distributed Systems and Operating Systems Group

Master Thesis

Specification and Design of a Reference Monitor Architecture Pattern in a Secure Operating System

Submitted by:
Eric Wintzler

Advisors: Prof. Dr. Boris Koldehofe
Dr.-Ing. Peter Amthor

Program: Informatik 2021
Matriculation Number: 58590

Submission Date: 17th May 2024

Contents

1 Introduction **1**
1.1 Problem Statement . . . 2
1.2 Goals of the Thesis . . . 3
1.3 Solution approach . . . 3

2 Fundamentals **5**
2.1 Access Control Enforcing Mechanisms . . . 5
2.1.1 Basic Definitions . . . 5
2.1.2 The Flask Architecture . . . 6
2.1.3 The Distributed Flask Architecture . . . 7
2.1.4 The Extended Flask Architecture . . . 7
2.1.5 The eXtended Access Control Markup Language . . . 9
2.1.6 The Policy Machine and Next Generation Access Control . . . 11
2.1.7 The AppSPEAR Framework . . . 14
2.2 Rust as a Language for reliable and secure System Software . . . 15
2.3 Modern Rust Operating Systems . . . 16
2.3.1 Tock . . . 16
2.3.2 Theseus OS . . . 18
2.3.3 Redox . . . 19
2.3.4 Poplar . . . 20

3 Requirements Analysis **21**
3.1 Methodology of the Requirements Analysis . . . 21
3.2 Requirements Analysis of Access Control Enforcing Mechanisms . . . 22
3.2.1 The Flask Architecture . . . 22
3.2.2 The Distributed Flask Architecture . . . 24
3.2.3 The Extended Flask Architecture . . . 25
3.2.4 The Architecture for the eXtended Access Control Markup Language . . . 27
3.2.5 The Policy Machine Architecture . . . 29
3.2.6 The AppSPEAR Architecture . . . 32
3.2.7 Comparison of the Access Control Enforcing Mechanisms . . . 33
3.3 Requirements Analysis of Access Control Policies . . . 35

4 Architecture Design **37**
4.1 Basic Components . . . 37
4.2 Refinement of Components and Interfaces . . . 40

5 Architecture Implementation **45**
5.1 RM Implementation Alternatives . . . 45
5.1.1 Operating System Environment . . . 45
5.1.2 Operating System Integration . . . 46

5.1.3 Internal Component Design 48
5.2 Details of the RM Implementation 50
5.2.1 Operating System Selection 51
5.2.2 Integration of the Reference Monitor 51
5.2.3 Internal Data Structures 51
5.2.4 Achieving policy flexibility 54
5.3 Remaining Tasks 54

6 Summary **55**

A Legends and Tables **57**
A.1 Explanation of Diagrams 57
A.1.1 Explanation of UML Component Diagrams 57
A.1.2 Explanation of UML Sequence Diagrams 58
A.2 Comparison of the Requirements of the AC Enforcing Mechanisms . 60

Bibliography **71**

CHAPTER 1

Introduction

Due to the large number of critical applications relying on a secure operating system, unauthorised access to the controlled resources should be prevented. Because of this a mechanism to enforce a security policy exists in many operating systems. A security policy is a set of rules that should help to guarantee a security objective: an abstract formulated property of the security of computer systems.

As a predominantly analysed subset of security policies used in operating systems, there exists Access Control policies (AC policies). According to [FGHK05] access control "is the administrative and automated process of defining and limiting which system users can perform which system operations on which system resources".

This definition implies the existence of a set of functional requirements that add security functionality – required for enforcing AC policies – to the inherent functions of an operating system. These functionalities are the mandatory and efficient enforcement of the policy, the secure and persistent storage of the policy and policy-related information as well as user authentication and policy administration.

In order for such a mechanism to achieve the desired effect, its implementation has to follow certain principles. These principles are described in [And72] as the concept of a Reference Monitor. This is an architectural component mandatory involved in any operation and protected against unauthorised manipulation. In addition, this component is sufficiently small and well-structured to be accessible to formal methods of correctness checking.

Consequently, on the non-functional side, two complementary and possibly conflicting non-functional requirements must be fulfilled, when implementing such a mechanism. On one hand the implementation should be as performant as possible to not influence the overall system performance in a functionally unacceptable way. On the other hand the implementation should be isolated in such a way that neither faults of the implementation can influence the operating system nor vice versa.

To address some of these goals, a programming language could be chosen, that provides performance as well as error isolation. Such a language is Rust [MK14]. Like C and C++, Rust enables the developer to take control over the memory usage of his code. Unlike C and C++, Rust not only knows about the syntax of memory usage, but also implements a semantics for this and thus enables memory usage free of errors. Due to these properties, Rust is also regarded as the preferred programming language for developing secure operating systems [BBB+17].

However, using Rust only reduces the effort for achieving fault isolation while preserving performance. With achieving fault isolation, also the requirement of the Reference Monitor being protected against unauthorised manipulation can be achieved. However, the use of Rust has no influence on the requirement of

the Reference Monitor being mandatory involved in any operation. Also, the requirement of the Reference Monitor being small enough and well-structured to be accessible to formal methods of correctness checking does not become less hard with the usage of Rust.

1.1 Problem Statement

This thesis deals with the question of a suitable architectural design for an operating system with mandatory enforced AC which fulfils the aforementioned requirements. For this, there exists many publications focus on developing an architecture for enforcing an AC policy in a software system. However, many of them are not applicable for arbitrary AC policies or operating systems.

On the other side, most of the proposed architectures have common elements: they all share a Policy Enforcement Point, a Policy Decision Point, a Policy Administration Point, a Policy Information Point and an Event Processing Point. However, this is the only common part of these publications, since they all define the interactions of these components and therefore their functionality in different ways.

With this the first problem for this thesis to solve is the question "How should the interfaces of a Reference Monitor architecture for an operating system AC be defined?"

The mentioned approaches for security-focused, AC-policy-controlled software systems however do not have their focus on enabling the enforcement of arbitrary AC policies. For example the Policy Machine described in [FGHK05] is more focused on abstracting arbitrary AC policies with the proposed, attribute-based NGAC models and how such an NGAC policy can be evaluated. Also, the XACML standard described in [OAS17] is more focused on defining a specification language for arbitrary AC policies than on defining how a mechanism for enforcing an AC policy could be implemented.

In addition, only a few of these publications are considering the environment – in particular an operating system – where their proposed architecture should be implemented. With this they leave out the problems coming with fulfilling the properties of a Reference Monitor – in particular the property of protecting against unauthorised manipulation.

Therefore, the second problem for this thesis to solve is the question "How can the Reference Monitor be implemented as independently as possible from the rest of the operating system kernel using a security enhancing programming language such as Rust?"

With implementing the Reference Monitor in an operating system, the third problem for this thesis to solve is the question "How can the trade-off between the non-functional requirements of operating systems and the Reference Monitor architecture being assessed?"

1.2 Goals of the Thesis

For solving the aforementioned problems, the goals of this thesis are

1. Develop a definition of the functional and non-functional requirements for the parts of an architecture for a Reference Monitor,
2. Develop the functional requirements for AC policies that are developed according to a state-of-the-art AC model, and
3. Develop a design specification of a Reference Monitor architecture pattern.

Thereby the results achieved by realising the first goal should serve as basis for a specification for realising the third goal by developing the design of the interfaces of a policy-driven operating system, assuming a modularised operating system architecture with address space isolation. Since the designed Reference Monitor should be able to enforce as many AC policies as possible, the results achieved by realising the second goal should also influence the design developed for realising the third goal.

The proposed architectural solution achieved by realising the third goal, should be capable of being implemented within a state-of-the-art operating system and support the enforcement of state-of-the-art AC policies. For evaluation purposes, this architecture is implemented within a highly modular operating system that is written in Rust and provides address space isolation. This implementation should be as strong as possible isolated from other parts of the operating system kernel.

1.3 Solution approach

For achieving the defined goals, in chapter 2 necessary fundamentals are surveyed. This includes AC enforcing mechanisms found in the literature and a motivation for using Rust as a language for reliable and secure system software. Also, an overview of modern operating systems written in Rust is given.

To define the functional requirements of the components of a Reference Monitor architecture and develop a specification out of them, in chapter 3 the components and interactions between these components are identified. On this basis the specification of these components is identified by analysing their input, output and conditions for both to be valid.

For developing the functional requirements for enforcing state-of-the-art AC policies, in chapter 3 the functional requirements of AC policies that are developed on the basis of a model calculus with a very high expressive power are analysed. This is based on what functionality is required for enforcing such a policy and how this functionality can be mapped to the specifications of the components developed before.

On the basis of these specifications, in chapter 4 the architecture of the Reference Monitor is specified. This is done by mapping components required by state-of-the-art AC policies to the individual properties of a Reference Monitor.

Based on this design specification, in chapter 5 a reference implementation is developed to validate the practical feasibility of the proposed architecture design.

For this at first the alternatives on how such an implementation could be done are discussed. On the basis of this discussion one alternative is chosen for being implemented. For this alternative implementation details are discussed.

Based on this work, chapter 6 summarises how the goals of this thesis were achieved. Also, an outlook on possible further work is given.

CHAPTER 2

Fundamentals

One goal of this thesis is to develop a definition of the functional and non-functional requirements of a Reference Monitor (RM) architecture that can handle the enforcement of state-of-the-art AC policies. Therefore, this chapter reviews literature on quality properties of RM architecture in an Operating System (OS). For this section 2.1 presents already existing solutions for that.

Another goal of this thesis is then to develop a specification and design of an RM architecture. For evaluating purposes, this design should be implemented into a state-of-the-art OS. Therefore, section 2.2 motivates the usage of state-of-the-art programming languages and tools that allow secure and reliable implementation of critical systems' software for OS development. Then section 2.3 selects and presents some recent OS written in Rust, which are analysed to determine what mechanisms they implement and provide that are useful for an RM implementation.

2.1 Access Control Enforcing Mechanisms

This section presents a review of the literature on mechanisms for enforcing an AC policy in a software system. Since an OS can be seen as a highly complex software system, all architectures designed for an arbitrary software system may also contain functionality that are needed in enforcing AC policies in an OS. This section thereby focuses on the architecture, the definitions of the architecture components and the environment in which the implementation of the proposals are considered.

2.1.1 Basic Definitions

Access Control policies in operating systems are usually specified in terms of subjects and objects. Thereby a *subject* is an active resource of the OS, i.e. a process that does something according to its users (a human) intention. An *object* is every passive OS resource that may be protected by an AC policy.

For enforcing an AC policy, information about the rights of the subjects on the objects must be stored somewhere. One widely used approach for that are *Access Control Lists (ACLs)* and *Capabilities*. An ACL is thereby a metadata component of an object that specifies the rights of subjects to access that object. In contrast, a capability is a metadata component of a subject that specifies the subject's rights to access specific objects.

2.1.2 The Flask Architecture

In [SSL+99] a security architecture for operating systems, which is called *Flask*, is presented. According to the authors of this publication, Flask is the first OS security architecture that is policy flexible. It is based on a modification of the Fluke microkernel but is also applicable to other operating systems as long as their kernel provides some form of process separation.

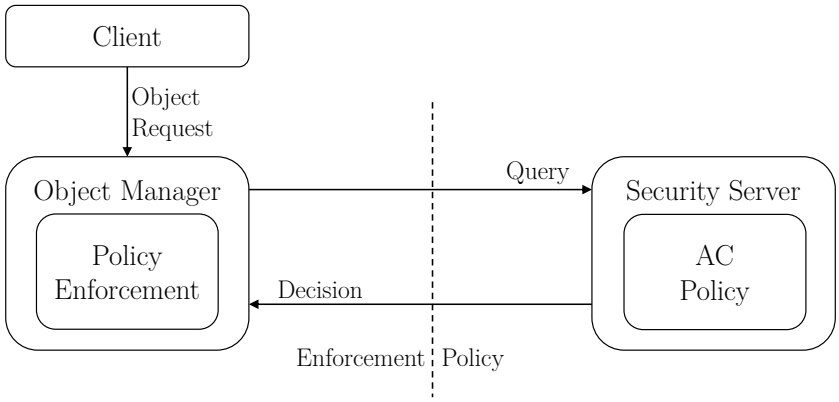


Figure 2.1: The elements of the Flask architecture and their interaction.

As shown in figure 2.1 the Flask architecture describes the interaction between object managers and a security server. The object managers are subsystems that are responsible for enforce policy decisions regarding their controlled objects against a client.

This architecture provides three components for object managers. The first one is an interface for receiving decisions of the security server in terms of Access Control, object labelling and poly-instantiation. A decision in terms of Access Control is thereby a specification whether – according to the AC policy – a permission on a specific object is granted to a subject that is requesting access to this object. A decision in terms of object labelling means that the AC policy assigns a label – a security identifier – to a controlled object according to its security attributes, which is used to encode these attributes for access requests of arbitrary policies. A decision in terms of poly-instantiation is a decision which specific instance of a poly-instantiated object a subject should access.

The second component of an object manager in the Flask architecture is a so-called access vector cache. This is a cache where the object manager holds AC decisions it has received from the security policy. By using the access vector cache the secondary objective of minimising the performance overhead can be achieved.

The third component of an object manager in the Flask architecture is an interface to receive notifications of policy changes. This is required because of the caching mechanism from element two. When first looking up the cache for an AC decision there is the risk of enforcing an old policy. A mechanism to signal that the cached decisions may be no longer valid is therefore required.

The security server is the subsystem that holds the AC policy and calculates AC decisions. It is also responsible for managing the mapping between security

identifiers and security attributes of the controlled objects. Also, the security server is responsible for managing the access vector caches of the object managers and enforcing the AC policy over its own services. Additionally, security server implementations may have a mechanism for loading and changing the policy.

One important point of the Flask architecture is, that a change of the security server implementation – e.g. to adopt it to a new kind of policy – does not require any changes of the object managers implementations. With this policy flexibility in terms of supporting arbitrary policies is achieved.

2.1.3 The Distributed Flask Architecture

In [PYG00] a Flask architecture for distributed systems was proposed, which will be referred to as *Distributed Flask (DFlask)* in this thesis. From the architectural side this proposal is very close to the Flask architecture presented in section 2.1.2: It consists of one Policy Decision Point (PDP; the AC policy in the security server in Flask) and one Policy Enforcement Point (PEP; an object manager in Flask).

In compliance with the Flask-Architecture the basic interaction between these components starts with the PEP which is requesting an AC decision from the PDP. The PDP then calculates the requested decision and sends it back to the requesting PEP.

In contrast to the Flask architecture the environment of this architecture is not an OS but a computer network with one or more nodes. With this the PEP is a component running on a network node and there is only one PEP at every node. Also, there does not have to be a PDP in every node. Rather, the PDP is a remote instance which may reside on a security server but can also be present locally. Additionally – and also in contrast to the Flask architecture – the PDP can contact other servers (for example an LDAP server) for calculating its decision or perform other services (for instance authentication) towards this.

When having a remote and a local PDP, the local PDP is used to decide the AC request based on the local policy configuration. The PEP then sends the AC request, the result and policy set of the local PDP to the network-located PDP. This one acts as the final authority for all AC requests and combines the decision of the local PDP and its own. The network-located PDP sends the final AC decision then back to the PEP.

2.1.4 The Extended Flask Architecture

In [PBP13] concepts and terms for systems security (in especial for policy based management) are presented. This includes an architecture for enforcing AC policies, which will be referred to as *Extended Flask (ExFlask)* in this thesis. In contrast to the architectures described in section 2.1.2 and section 2.1.3 this architecture consists of five elements, as shown in figure 2.2: a Policy Enforcement Point (PEP), Policy Decision Point (PDP), Policy Information Point (PIP), Policy Administration Point (PAP) and a Policy Repository.

The PAP is used for managing the policies that are stored in the Policy Repository. For this the PAP provides a user interface to create and store a new policy and to

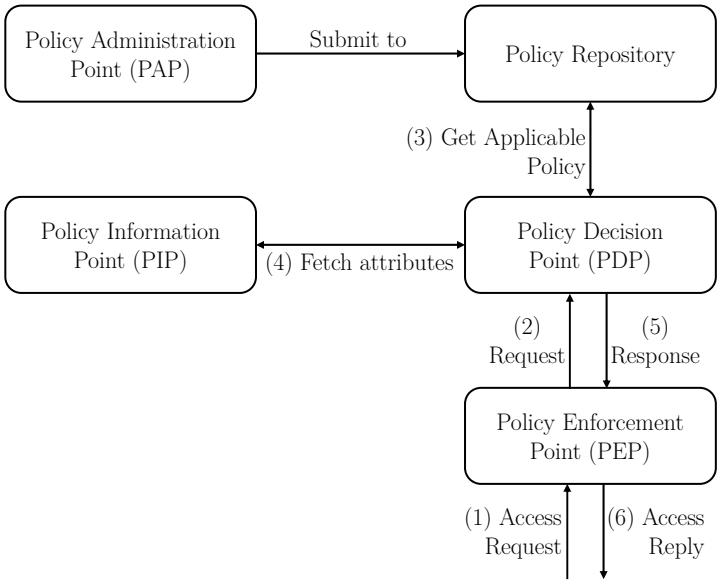


Figure 2.2: The elements of the policy-driven system management architecture and their interaction on an access request.

validate or transform an existing policy. With having a policy repository there is one main difference between this architecture and the Flask or DFlask architecture. The architecture presented in this section is specified with the possibility to enforce multiple policies at the same time.

Since there can be multiple policies the first functionality of the PDP is to identify which policies are applicable for a given AC request. After that the functionality of the PDP is the same as for the PDP in the DFlask and the security server in Flask: it checks against the policy(ies) if the given AC request must be allowed or denied. While doing this, the PDP has also to resolve possible conflicts between the applicable policies.

Because some policies require additional information to the raw AC request (for instance attributes of the subject or object), such information must be stored somewhere. In the Flask architecture this is done by the security server in terms of object labelling (but only for objects) and in DFlask by the use of external services. In the architecture presented in this section this information are stored in the PIP. The PIP can be seen as a database which stores all kind of context information requires for the evaluation of the policy(ies) by the PDP.

The PEP then serves as a user interface to the policy. It gets an AC request from the subject and enforces the decision of the PDP against this subject. With this its functionality is similar to the object manager in the Flask or the PEP in the DFlask architecture.

The process for executing an operation in this architecture is now as shown in figure 2.2. At the beginning the PEP receives an AC request from the subject that wants to perform an operation on an object. The PEP sends this request to PDP. The PDP then searches for applicable policies in the policy repository and fetches the required attributes from the PIP. With this information the PDP calculates the policy decision and sends it back to the PEP. The PEP then enforces this decision against the subject by rejecting the request or perform the requested operation.

2.1.5 The eXtended Access Control Markup Language

In [OAS17] a specification language for Attribute Based AC policies (ABAC policies) based on XML, the *eXtended Access Control Markup Language (XACML)* is specified. Next to specification of the language there is also a specification on how to evaluate policies that are expressed in XACML. In preparing the specification of the policy evaluation a data flow model is presented that can directly mapped to an architecture for enforcing AC policies.

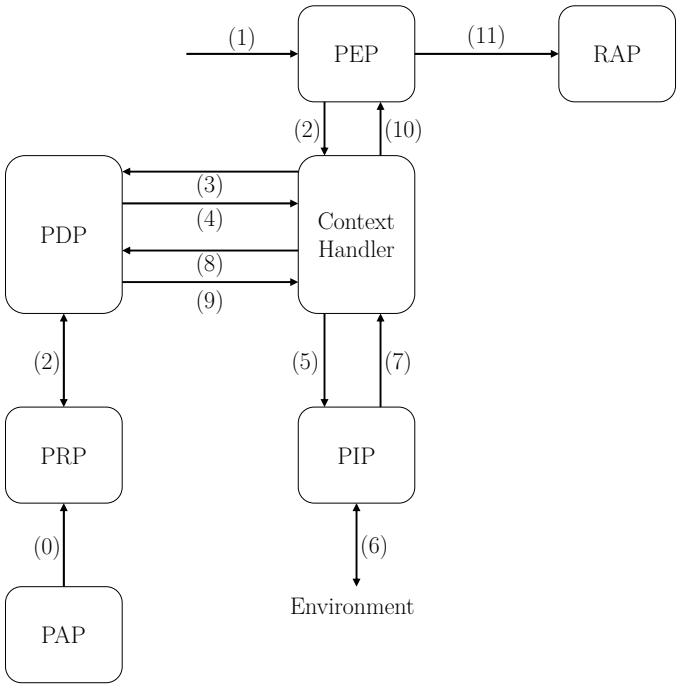


Figure 2.3: The simplified XACML enforcement architecture.

This architecture is outlined in figure 2.3 and consists of eight elements: a Policy Administration Point (PAP), a Policy Retrieval Point (PRP), a Policy Decision Point (PDP), a Policy Information Point (PIP), a Resource Access Point (RAP), a context handler and the Policy Enforcement Points (PEPs). However, some of these components – for instance the PRP have empty interfaces from a functional point of view. This means that they did not implement any logic but a direct forwarding of their stored information.

While PDP, PAP, PEPs, PIP and the context handler are specified in [OAS17], the PRP and the RAP are not. These were taken from [FCHK16], where the XACML standard is compared to the Policy Machine approach which is described in section 2.1.6. Thereby the PRP can be seen as a place where the policies are stored – like the policy repository in ExFlask from section 2.1.4. However, the PRP is implicitly mentioned in [OAS17] by stating that some presented data flows – for instance between the PAP and the PDP – "may be facilitated by a repository".

In contrast to the PRP, the RAP is an abstraction of an existing component. According to [OAS17], obligations can be specified that have to be fulfilled from the PEP. However, this term *obligation* is only defined as "an operation specified

in a rule, policy or policy set that should be performed by the PEP in conjunction with the enforcement of an authorisation decision". Because of that this thesis assumes that this means that the operation is performed on a systems' resource. For accessing these resources the RAP is responsible.

By comparing this architecture to the ExFlask architecture, most of the components not explained yet have the same functionality: The PAP can be used for creating or modifying the policies stores in the PRP. The PDP is responsible for calculating the policy decisions by querying the policies stored in the PRP and using additional information that are not stored in the policy. The PIP serves as retrieving point for this additional information. The PEPs are responsible for allowing or denying access request by querying the PDP.

In contrast to the ExFlask architecture, the communication between the PDP, PIP and the PEPs is not direct. Instead, all of this communication is done via the context handler. According to [FCHK16] this is done, because the PEP is integrated into an operating environment, which is tightly coupled to its application. Because of this the PEP might not formulate its request according to the XACML protocol, and the context handler servers as a translator between the PEP and the PDP. Also, the context handler retrieves the required information for the PDP from the PIP.

Another contrast to the ExFlask architecture is, that the PIP is just an indirection for getting the required information. As shown in the data flow model in [OAS17], the PIP just collects the requested information from the environment. This includes information about the subjects and resources.

In comparison to the ExFlask architecture, the process for an AC request is now as shown in figure 2.3. This includes another difference: unlike in the ExFlask architecture, the XACML architecture does not specify any response to the subject that has requested access to a controlled object.

(0) Before any operation can take place, a policy must be installed via the PAP.

(1) The PEP receives an AC request from a subject for a specified operation on an object.

(2) The PEP forwards the AC request in its native format to the context handler. Optionally the PEP includes the attributes of the entities involved in this request into the request.

(3) The context handler constructs the *XACML request context* from the request and sends it to the PDP. Optionally the context handler includes the received attributes into this request.

(4) The PDP requests any additional required attributes of the subject, object, action or environment from the context handler.

(5) The context handler requests this attributes from the PIP.

(6) The PIP obtains the requested attributes from the environment.

(7) The PIP sends the obtained attributes back to the context handler.

(8) The context handler sends the retrieved information back to the PDP.

(9) The PDP sends its decision of the AC request (the *XACML response context*) to the context handler.

(10) The context handler translates the response context into the native response format of the PEP and sends the response to the PEP.

(11) The PEP enforces the decision of the PDP and fulfil its obligations.

When comparing this architecture to the Flask and DFlask architecture one difference is the number of components. The Flask and DFlask architectures both specify only two components: a set of object managers (PEPs) and a security server (PDP).

Thereby the functionality of the object managers in Flask and the PEPs in DFlask is comparable to the functionality of the PEPs in the XACML architecture: they both getting an AC request from a subject and enforce the decision of the security server (PDP) against this subject.

In contrast to the security server in the Flask and the PDP in DFlask architecture the PDP in this architecture has a reduced functionality. It is only responsible for calculating if an AC request has to be allowed or denied. It does not maintain any information about the systems resources and does not provide methods for changing and loading the policy(ies). These functionalities are outsourced into the PIP and PAP.

Because the XACML architecture allows – in contrast to the Flask and DFlask architecture – the storage and enforcement of multiple policies, the functionality of the PDP is extended such that multiple policies may be combined in a deterministic way. This is done in [OAS17] by defining a set of combination algorithms. In contrast to the Flask and DFlask architecture this specifies parts of the enforcement logic in the PDP. For doing so, the XACML architecture limits the types of policies that can be enforced to those that can be expressed in XACML. This is a contrast to the Flask and DFlask architecture, which are designed with the objective of not limiting the types of policies that can be enforced.

2.1.6 The Policy Machine and Next Generation Access Control

In [FGHK05, FAG11, FGJ14, FCHK16] an approach for abstracting ABAC policies is specified. This approach was developed under the name *Policy Machine (PM)* and later specified as an ANSI/INCITS standard under the name *Next Generation Access Control (NGAC)*. While [FGHK05] focuses on how to abstract arbitrary ABAC policies by defining an ABAC model, in [FAG11, FGJ14, FCHK16] also an architecture for enforcing policies developed against such models is specified.

This architecture is composed out of six components and shown in figure 2.4. These components are a set of Policy Enforcement Points (PEPs) in combination with a set of Resource Access Points (RAPs), a Policy Decision Point (PDP), a Policy Administration Point (PAP), a Policy Information Point (PIP) and an Event Processing Point (EPP).

By comparing this architecture to the XACML architecture as described in section 2.1.5 and the ExFlask architecture, the PIP, PAP, PDP, PEPs and RAPs

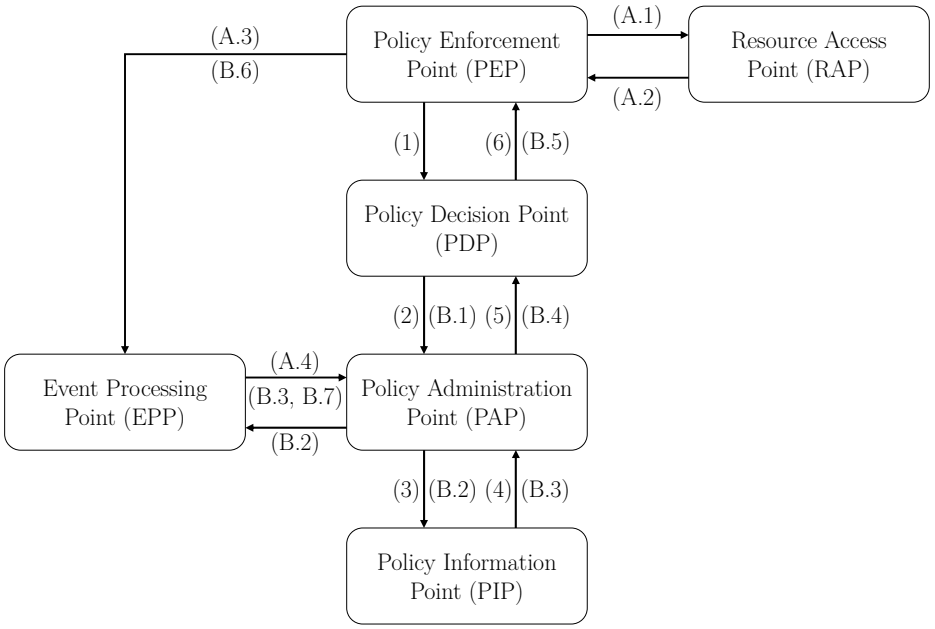


Figure 2.4: The Policy Machine security architecture.

have nearly the same functionality. However, there are differences in how they are implemented in detail.

Comparing to the XACML architecture the first difference is that the PM architecture does not specify a context handler and therefore all communication between the components is done directly. The second difference is the missing PRP. In contrast to the XACML architecture, in the PM architecture it is assumed, that the policies are stored directly in the PDP. The third difference is that the PM architecture defines an Event Processing Point (EPP). This component is invoked whenever an AC request was permitted and can do policy modifications via the PAP. The EPP thereby has a similar functionality as the obligations in the XACML architecture. The fourth difference is the placement of the PAP. While in the XACML architecture the PAP is placed after the PRP and can therefore only modify the AC policy, in the PM architecture the PAP is placed between the PDP, PAP and PIP. With this the PDP has to invoke the PAP for getting information from the PIP but can also modify the state of the policy, which is also possible for the EPP with this.

In accordance with the XACML architecture, the PM architecture defines the functionality of the PEPs. They receive an AC request from a subject and enforce the decision of the PDP against it. However, in detail there is a difference: The PEPs are now also enforcing request for operations on the PAP – so-called *administrative operations* – but also checking the result of an operation on the respective RAP. Also in accordance with the XACML architecture is the functionality of the RAPs: they serve as access points to the controlled resources.

In comparison to the XACML and ExFlask architecture the process of enforcing an access request is now as shown in figure 2.4:

(0) A PEP receives an AC request for a specified object and operation from a subject. This request includes also an identifier of the user in which name the subject acts. (Not displayed)

(1) The PEP sends this request to the PDP.

(2) The PDP queries the PAP to look up the required information.

(3) The PAP requests for the requested information on the PIP.

(4) The PIP sends the requested information back to the PAP.

(5) The PAP sends the collected information from the PIP to the PDP.

(6) The PDP calculates if the AC request has to be allowed or denied.

From this point there are two different ways of how to go on, depending on whether the operation in the AC request is an administrative operation or not. If the operation is not an administrative operation path A will be used, otherwise path B.

For a non-administrative operation, path A is performing the operation on the object via the RAP (step A.1) and checks the result of that (step A.2). If the execution was successful, the context of the AC request will be sent to the EPP (step A.3) which triggers the events that matches this context (step A.4). These events may alter the policy state, by changing the information in the PIP or EPP.

For an administrative operation, on path B the following steps are performed:

(B.1) The PDP sends the command to execute the administrative operation on the PAP.

(B.2) The PAP executes the administrative operation. This is done by altering the state of the PIP or EPP.

(B.3) The status of the execution of the PIP or EPP is returned to the PAP.

(B.4) The PAP returns the status of the execution of the administrative operation to the PDP.

(B.5) The PDP forwards this state to the PEP.

(B.6) If the execution was successful the PEP sends the context of the AC request to the EPP.

(B.7) If an event trigger matches to the AC context, the actions defined by this event are executed via the PAP. The PAP changes the information stored in the PIP or EPP according to the defined actions.

When comparing the PM architecture to the ExFlask architecture there are some similarities. Both architectures specify a set of PEPs, a PDP, a PAP and a PIP. In both architectures the PEPs are defined to receive an AC request, send it to the PDP and enforce the decision of the PDP. Also, the functionality of the PDP is similar in both architectures: the PDP receives an AC request from a PEP, looks for additional information at the PIP and is influenced by the PAP.

Comparing the PM architecture to the ExFlask architecture, differences can be found first in terms of the specified components and their placement. This especially holds for the PAP. While in the ExFlask architecture the PAP is only

responsible for modifying the policies stored in the policy repository and therefore only connected to this, in the PM architecture the PAP is also responsible for modifying the information stored in the PIP and the events stored in the EPP. With this a difference in the functionality of the PIP can be found, since in the PM architecture the information stored in the PIP can be modified, which is not possible in the ExFlask architecture. Another difference is that the ExFlask architecture does not have an EPP. In particular the ExFlask architecture does not specify any mechanism to perform actions after an AC request was allowed. The next difference is the explicit occurrence of the policy repository. In the PM architecture this is moved into the PDP. The last difference is that the PEPs are not given any reply to the subject which has triggered the AC request.

By comparing the PM architecture with the DFlask and the Flask architecture the similarities and differences are nearly the same as described for the XACML architecture in section 2.1.5. In contrast to the Flask and DFlask architecture, the PM architecture defines also a point that has a special logic for processing AC enforcement related events: the EPP.

2.1.7 The AppSPEAR Framework

In [Sch21] AppSPEAR, a framework for enforcing AC policies in an application software is presented. With this it is designed for a different domain. However, the semantics of application AC policies includes a subset of functionalities that are also needed for enforcing AC policies on OS level. In addition, AppSPEAR shows how the terms used in the previous sections can be used in the domain of application software.

As shown in figure 2.5 AppSPEAR consists of three main elements: a set of Trusted Object Managers (TOMs), one Trusted Policy Server (TPS) and one Trusted Event Processor (TEP). In addition, AppSPEAR contains some kind of policy storage.

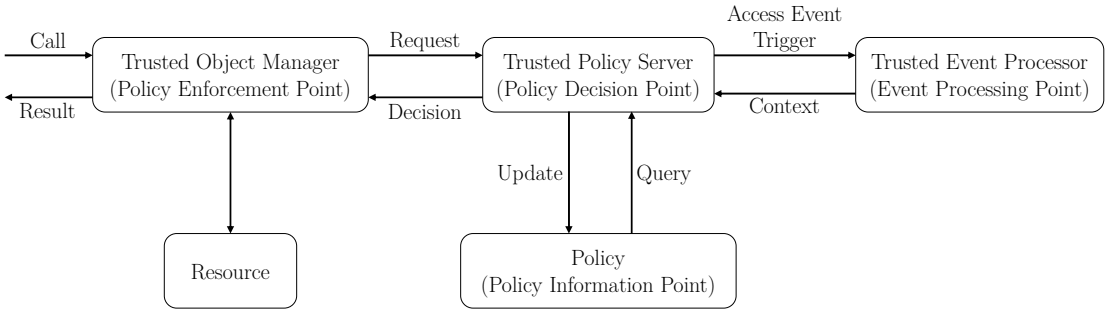


Figure 2.5: The components of the AppSPEAR framework and their interaction.

In AppSPEAR the TOMs are responsible for enforcing the policy. This includes that they are involved in every operation on the controlled objects and querying the TPS for calculating a decision if a given access should be allowed or denied. If access is allowed by the TPS, the respective TOM also performs the requested operation. With having this functionality the TOMs can be seen as Policy Enforcement Points (PEPs).

The TPS is the runtime environment of the AC policy. It calculates whether an AC request given from a TOM has to be allowed or denied. In addition, it is also responsible for a persistent and secure storage of the AC policy. With this functionality the TPS is comparable to a Policy Decision Point (PDP). However, as shown in [Sch21, figure 1] the task of storing the policy may be outsourced to another component with may be referred as Policy Information Point (PIP).

The TEP is an optional add-on for providing context based security policies information about the context of the system. Since this information may change with allowing an AC request, the TPS is able to interact with the TEP in a bidirectional manner. For performance reasons it is supposed to implement the invocation of the TEP through event triggers. With having this functionality the TEP can be compared to the EPP of the PM architecture.

When comparing the AppSPEAR architecture to the PM, XACML and ExFlask architecture the first thing to note is the missing of a PAP in AppSPEAR. For this [Sch21] stated that a PAP is omitted since the publication focuses on enforcement and therefore not specifies a PAP as entry point for administration.

The second difference to these architectures is the specification of the PIP. While in the XACML and PM architecture a PIP is explicitly specified for serving as lookup resource for additional information required for policy decisions, in the AppSPEAR architecture a PIP is only implicit mentioned in [Sch21, figure 1] and seems to serve as place to store the policy itself – a functionality comparable to the PRP in the XACML architecture or the policy repository in the ExFlask architecture.

Comparing the AppSPEAR architecture to the Flask and DFlask architecture, the biggest difference is the existence of the TEP in AppSPEAR which is not specified for Flask or DFlask. Another difference is the existence of the Policy Storage (PIP) in AppSPEAR, which is not specified for the Flask and DFlask architecture.

2.2 Rust as a Language for reliable and secure System Software

Typically, operating systems and other system software is written in C, a language that is deliberately close to the underlying hardware in terms of both provided abstractions and guaranteed non-functional properties. Since C is, according to [PCO19], designed to give the developers a huge amount of freedom on how to achieve their goals, it is hard to make any guarantees about the code on runtime. Today there exists a lot of languages, with some of them (like Ada) are designed for the domain of safety-critical software. But, according to [BBB+17], traditional safe languages rely on a managed runtime and specify garbage collection to implement safety – a huge overhead for nowadays operating systems.

In the last years a new programming language was specified which, according to [PCO19], promises reliability in combination with high performance. This language is the Rust programming language [MK14]. In detail – according to [BBB+17] – Rust provides type and memory safety via an ownership model, where a unique

reference to each valid object exists. Through this, every object can be tracked statically and there is no need to handle object deallocation with the help of a garbage collector. With this, code written in Rust can be compiled into native machine code, which is a strong point in terms of performance.

This ownership model in combination with Rusts lifetime model allows for generating actions for object deallocation at compile time. Through this Rust allows a flexible memory management with making less errors than doing this manual. Since with this Rust not only know about the syntax of memory accesses (like addresses in C), but also a semantics (Rust asks whether the referred value is still valid), the compiler is able to perform a static code analysis.

According to [PCO19] Rust also can be used for "fearless concurrency with the help of a number of abstractions and safety guarantees". Also, Rust can be seen as a portable language, since there exists an embedded-hardware abstraction layer and the Rust toolchain is available for a wide range of platforms.

However, if Rusts safety mechanisms are a barrier for the software to write – for example when a direct register access is necessary – these mechanisms, more precisely the static analysis through the compiler, can be disabled for marked code blocks. These code blocks are then using unsafe Rust.

With having these powerful mechanisms of Rust, this programming language can be a useful tool for designing an RM architecture in an OS kernel. Because of Rusts compile time memory management and the compiler being able to perform a static code analysis, it can be easily verified that the RM is tamper proofed. With having this memory management also a formal verification of the RM becomes less hard. With having these compile time analysis, a correctness proof of a system becomes – according to [IBZ23] – less hard.

2.3 Modern Rust Operating Systems

This section gives an overview about state-of-the-art operating systems. This overview should later serve as basis for the decision which of these operating systems will be chosen for implementing an RM architecture into it. Because of this goal only operating systems were chosen which provides functionality that could be the target of a security policy.

For enabling such an analysis this section gives background information about these operating systems. This includes at first an analysis on the overall architecture of the operating systems. At second, it is analysed what security features the operating systems provides and if there is already implemented some form of an RM architecture.

Because of the requirement of an RM to be protected from unauthorised changes, it is analysed which features in terms of robustness these operating systems provide. With this it is also analysed how modular these operating systems are.

2.3.1 Tock

Tock is a reliable operating system for low power microcontrollers, presented in [LCG+17]. As shown in figure 2.6 [LSA+, section 4.1], it consists of two layers which

can be seen as kernel and user space. The kernel space is thereby divided into three component layers: capsules, the core kernel and peripheral drivers. The user space consists of processes, which can be compared to processes in a general-purpose OS.

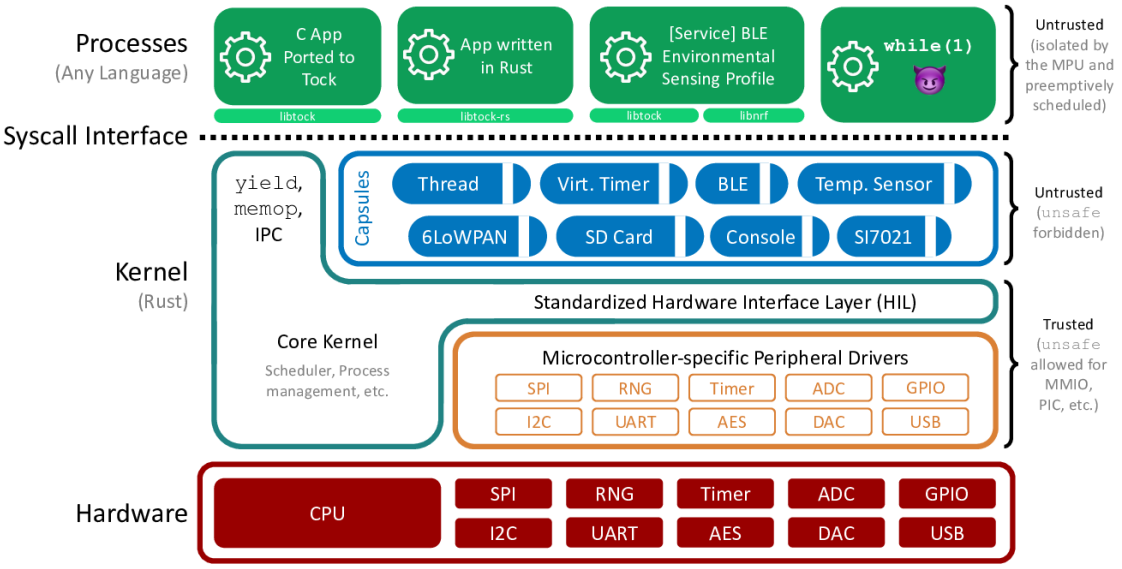


Figure 2.6: Architecture of the Tock operating system.

In the kernel space the peripheral drivers have direct control over the hardware and therefore can contain unsafe Rust. Because of this they are assumed as trustworthy. Between the peripheral drivers and the capsules lies the core kernel. This is the actual Tock kernel responsible for scheduling, process management, inter process communication and more. Because it must, e.g. for scheduling, write direct into hardware registers, the core kernel is also assumed to be trustworthy. The core kernel also exposes a standardised hardware interface layer to the capsules one layer above.

The capsules are defined as “units of composition within the kernel” that are scheduled cooperatively. They are instances of a Rust struct. With the core kernel providing the hardware interface layer, they do not need direct access to the hardware and must not be trusted, which means that they must not contain blocks of unsafe Rust code. Capsules are isolated from each other by Rusts type and module system, so no capsule can access data or memory of another capsule.

Using Rust for isolating the capsules have the advantage, that the isolation can be done at compile time and there are no runtime actions required. This leads to minimal memory usage at runtime for a fine-grained isolation mechanism, that protect the kernel against errors. This isolation is a point in terms of robustness, because of the cooperative scheduling of the capsules.

The processes are “hardware-isolated concurrent executions of programs, similar to processes in other systems”, which are isolated by hardware. This is a stronger isolation than the language based sandbox for the capsules. Because of this isolation, malicious processes – e.g. an infinite loop – cannot influence the system. Because of Tock providing a hardware abstraction in terms of a system call interface, processes can be written in any language.

Also in contrast to the capsules, processes have their own logical memory region with its own stack, heap and static variables. This allows Tock to schedule processes with a pre-emptive round-robin strategy. But in contrast to general purpose operating systems, Tocks processes have no illusion of infinite memory.

2.3.2 Theseus OS

Theseus OS is an OS written in Rust that is presented in [BLIZ20]. The intent of writing this OS was to show how *state spill* can be minimised in an OS. According to [BVZ17], state spill is "the act of one software entity's state undergoing lasting changes as a result of its interaction with another entity". With this Theseus focusses on enforcing a non-interference policy, like defined in [GM82]. For achieving the goal of reducing state spill – as stated in [BLIZ20] – Theseus OS follows a novel structure of many small components which have clear defined and runtime persistent bounds.

These components are named *cells* and can be composed and exchanged at runtime. With this a cell provides a software defined unit of modularity, which serves a core module of the OS. A cell is a Rust crate at implementation time, an object file at compile time and a set of loaded memory regions with bounds per section at runtime. These explicit identifiable cell bounds are the foundation of a strong data and fault isolation in Theseus.

To achieve safety in Theseus, the opportunities of the Rust language and compiler are utilised as far as possible. For this the authors used a so-called *intralingual* approach, which is presented in [IBZ23]. With this the compiler is able to check safety and correctness invariants without gaps in its understanding of the behaviour of the code.

When using the intralingual design the compiler is also responsible for the resource management of the OS. With using the compiler for this, the states, that the OS must maintain, are reduced. Because of this, now state spill in the OS is reduced and isolation is strengthened. This is also a point in terms of safety because with this, semantic errors are shifted from runtime failures to compile time errors.

Because of the compiler can not check the behaviour in terms of interaction of independent compiled programs, the OS runtime environment is tailored to Rusts runtime model: all programs (including the OS) are running in a single address space and at a single privilege level, so that there is no other world or mode of execution.

To achieve safety in terms of robustness, in Theseus all cleaning up semantics of resources is implemented in the drop handler of a resource, so that the management of allocation and deallocation is shifted from runtime to the compiler. In addition to this Theseus also implemented a feature named *stack unwinding*, which ensures that a resource held by a process is freed in case of a normal or exceptional termination of the process. When applying this to locks a process may hold, many cases of deadlocks can be prevented. To not influence the systems overall performance, unwinding is only applied when required. With this there is no interference with the normal execution performance like with using a garbage collector.

Also, to achieve robustness in terms of availability, in Theseus exists three stages of failure handling. The first stage is to clean up the failed task via unwinding, so that tasks that share resources with the failed task can continue working. The second stage is to respawn a new instance of the failed task via an intralingual approach of respawnable tasks. The third stage is to use the mechanism of cell swapping to replace the failed tasks instances with freshly loaded ones at different memory locations.

To achieve the goal of reducing state spill, Theseus uses an approach called *opaque exportation* for state managing. This means, that in a client-server-interaction every client is the owner of the state that represents its progress with the server (exportation), but it can not arbitrarily access or modify this state because it is server private, ensured by Rusts type safety property.

2.3.3 Redox

The Redox operating system [Red] is a general purpose operating system written in Rust. It is a Unix like microkernel OS that should be secure and reliable. It is compatible to the POSIX standard so that it can run many programs that are developed for e.g. Linux.

Because of the microkernel design, drivers and many system services can run in user mode. With this they are similar to normal user programs and the kernel can restrict them to have only access to such resources they need for their task. If now such a driver or service fails it does not affect the whole kernel. Additionally, there is no need to react on such a failure in any special way. It is sufficient to restart the failed component like doing for a failed user process.

Also, because of its microkernel design, Redox is very modular. Since most system components are running in user space, they can simply be replaced, modified or removed. Many monolithic kernels try to achieve this by introducing kernel modules but often adding, changing or removing a module requires at least a system reboot.

Another point of the microkernel architecture is, that with such an architecture every of the drivers and system services that run in the user mode, run in their own address space. With this a failure of one component can not affect other components. While this is a point in terms of isolation, it has a strong impact on the systems' performance. This is because of the frequency with which context switches have to be performed and the overhead that is associated with every context switch.

Context switches have to be performed more often than in a monolithic kernel because of the functionality of system calls. In a monolithic kernel a system call invokes a kernel routine from user space and sends the result of this routine back to the user space. In the Redox kernel a system call also at first invokes a kernel routine. However, in contrast to such a routine in a monolithic kernel, the invoked routine in Redox sends a request to the appropriate driver or system service at user space. This user space driver or service then sends its result back to the kernel routine which sends it back to the original caller.

Redox comes with two security features. The first one feature of Redox is a "capability-based system". Thereby the term capability is used for customised file descriptor, but no other information about this feature were provided in [Red].

The second security feature is a namespace system, where each process is assigned to a namespace. In Redox a namespace is a list of schemes where a scheme is the type of the resources. Whenever a scheme is listed in the programs' namespace, the program is allowed to interact with the resources represented by this scheme.

2.3.4 Poplar

Poplar [Woo] is a microkernel operating system designed for research purposes that is designed to support a minimal system call interface and first class support for message passing based communication between user space processes.

Because of its microkernel architecture, Poplar has only a few system calls. These system calls are all based around so-called *kernel objects* – Polars representations of resources that can be interacted with from user space. This includes logging, creation and interaction with channels for Inter Process Communication (IPC), interaction with services, retrieving information about the hardware and memory requests. Every other functionality an application may need is provided by user space processes to which it is to communication via IPC.

In contrast to Redox, the Poplar kernel is reduced in its functionality. In Redox all system calls are handled in schemes and all operations that are using schemes are at first passed to the kernel which then pass it to the application responsible for that scheme. With this the Redox kernel implements a more complex functionality than the Poplar kernel, that does not know anything about schemes or its user space applications.

As security feature Poplar provides capabilities. These capabilities are encoded in the ELF image of a task. Since Poplar is designed as a microkernel, there are only a few capabilities defined because the kernel does not know about any operations on resources that are defined by user space processes.

CHAPTER 3

Requirements Analysis

One goal of this thesis is to develop the functional and non-functional requirements of the components of an RM architecture. For doing so, the first step is to define the methodology for this analysis as in section 3.1. Then, in section 3.2, the requirements of the AC policy enforcing mechanisms presented in section 2.1 are analysed according to the presented methodology.

For this thesis' goal to develop the functional requirements of AC policies, that are developed according to a state-of-the-art AC model, section 3.3 at first describes how such policies can be modelled. Based on this the requirements are analysed.

3.1 Methodology of the Requirements Analysis

For identifying the functional and non-functional requirements of the components of the presented AC policy enforcing mechanisms, the only information provided in the literature are basic interaction schemes like presented in section 2.1. However, requirements can be derived out of this schemes in terms of what is the input and the output of such an interaction and what properties need to hold for both.

With having the input and the output and the semantics of an interaction, there can be also derived what are the requirements for such an interaction to be valid and also the requirements that have to be fulfilled when such an interaction was successful. By doing so, the analyses following the approach of *Design by Contract* as described in [Mey92]. While Design by Contract is focussing on developing software in an object-oriented programming language, its method of identifying input, output and invariants of an operation as well as identifying conditions on the input and output for specific components can be used for the analysis of – possibly not object-oriented – software.

Applying this method to the AC enforcing mechanisms is done by modelling the interaction between two components as function calls and analysing what parameters are passed to these functions and which values they provide as output. Based on this it is analysed, what requirements the caller must fulfil to be correct and what requirements the called components fulfil after a successful execution of a call.

By doing this analyses the terms *precondition* and *postcondition* are used to describe the requirements that the caller must fulfil for a valid call or the requirements that the callee must fulfil after a successful execution. In this thesis the definitions from definition 3.1 for pre- and postconditions are used. These definitions were adopted from [MM02] to the described method of analysing AC enforcing mechanisms.

Definition 3.1 (Pre- and Postconditions).
A *precondition* is every condition that the caller of a function must fulfil to let the call be valid. This includes conditions on the input parameters as well as conditions that the caller – as a stateful component – must fulfil before the function is called.

A *postcondition* of a function is every condition that must be true after the successful execution of this function. This includes conditions on the output of the method as well as conditions that the callee of this function – as a stateful component – must fulfil after the return of this function.

3.2 Requirements Analysis of Access Control Enforcing Mechanisms

This section describes the analysis of the AC policy enforcing mechanisms from section 2.1. This is done for each of these mechanisms by listing all components of this mechanism. For each component then the possible interactions with other components are modelled as function calls. Each of this function calls is then analysed in terms of input parameters and output value(s) as well as its pre- and postconditions according to the methodology described in section 3.1.

For this, every enforcing mechanism is depicted as UML component diagram – see section A.1.1 for a legend –, where it is decomposed in its building blocks and their interfaces. Then the interactions between these building blocks are analysed as described. If thereby noting else is stated, the term *access request* is used as defined in definition 3.2.

Definition 3.2 (Access Request).
An access request is the action of a subject performing an operation on an object. It can be modelled as a tuple $\langle s, o, op \rangle$ where

- s is the identifier of the subject,
- o is the identifier of the object, and
- op is the identifier of the operation.

3.2.1 The Flask Architecture

The Flask architecture presented in section 2.1.2 consists of a set of object managers and a security server. Thereby the components of this architecture and their interfaces can be modelled as depicted in figure 3.1.

The Object Managers. Each object manager can be seen as a composition of three parts: an interface for policy interactions, a cache and a callback interface for policy changes. Thereby the part of the functional specification of an object manager responsible for the policy interactions can be seen as composed of three parts: one part responsible for dealing with policy decisions, one part responsible for object labelling and one part responsible for poly-instantiation decisions.

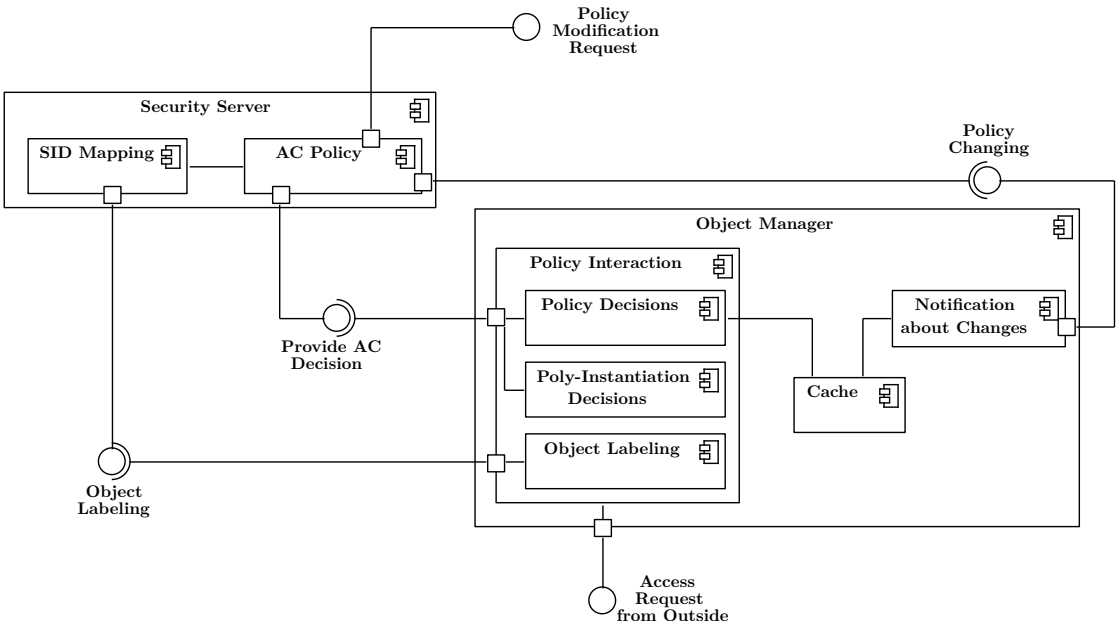


Figure 3.1: The components and interfaces of the Flask architecture.

However, when looking at the interactions between an object manager and the security server, two of the three parts for policy interactions can be modelled together in a single function call. These two parts are the part responsible for dealing with policy decisions and the part responsible for poly-instantiation decisions. This call is triggered at an object manager by a subject and gets an access request as input. When the call returns, it outputs the decision of the security server, whether the access request is allowed according to the AC policy and which instance of the object to use. Since this function call is triggered when a subject tries to access an object, there are no preconditions here. As postcondition the object manager enforces the decision of the security server. This is done by executing the requested operation only if the security server returns an allowing for that. In this case the execution is done at the instance given from the security server.

When giving an access request to the security server, the objects' metadata must be passed to the security server. This is done by first encoding the security relevant metadata as string – the so-called *security context* – and then encode this string as a fixed-size value, the so-called *security identifier (SID)*. This SID is assigned to a specific security context by the security server. The assignment must therefore be done whenever the (security relevant) metadata of an object changes or a new object is created. Because of this, the assignment process is triggered by an object manager and can be modelled as a function call on an object manager. The input value of this call is therefore the security context of the object and the output the calculated SID. For this call to be valid, the access request for changing the objects' metadata (or to create the specified object) must be allowed by the AC policy as a precondition. The postcondition is then that the SID given from the security server is assigned to the object. Because of [SSL+99] stating, that "the SID mapping cannot be assumed to be consistent across executions (reboots) of the security server nor across security servers on different nodes", there is no need for doing this assignment persistently.

Because the Flask architecture defines a cache for policy decisions at the object mangers, it is necessary to inform the object managers if the policy has changed and therefore the cached decisions may no longer be valid. This can be modelled as function call at the object managers, initiated by the security server. For a valid execution of this function, a change on the security servers policy must have occurred. Because this function is only to notify the object managers that a change in the AC policy has occurred, an input value is not necessary. After a successful execution of this function, each object manager have invalidated its cache and outputs an indicator of finishing this to the security server, to let it know from when all object managers have invalidated their cache and are enforcing the new policy.

The Security Server. The security server is responsible for making decisions whether a specific request has to be allowed or denied according to the stored AC policy. Because of having the same input and fitting to the workflow of the object managers, the second functionality of the security server – making decisions about poly instantiation – will be modelled as part of this function call. For this the security server gets the access request from the requesting object manager. This function then outputs the decision calculated from the AC policy – if the access is allowed and what instance of the object to use – to the object manager as output value.

The second responsibility of the security server is to maintain the mapping between security contexts and SIDs. This especially includes the calculation of an SID for a given security context. This functionality can be modelled as a function called from an object manager. The input parameter of this function is a security context. The output of this function is the calculated SID.

The last responsibility of the security server is to react upon a request of changing the stored AC policy. This can be modelled as a function call from outside to the security server. To let such a call be valid, the requested change must be allowed according to the currently installed AC policy as precondition. The processing of this request is then accepting the new AC policy to store as input and outputs an indicator about success or error to the caller. When the storage of the new AC policy was allowed by the current one and was successfully done, the security server informs the object managers about that change as postcondition.

3.2.2 The Distributed Flask Architecture

The DFlask Architecture presented in section 2.1.3 consists of one PEP per node of a computer network and at least one PDP. The components of this architecture and their interfaces can be modelled as depicted in figure 3.2.

The Policy Enforcement Point. The PEP is responsible for intercepting access requests and enforcing the decisions of the PDP. This can be modelled as a function call of a subject, trying to access an object. This function gets the access request as input and provides the decision of the PDP as output. Because of DFlask is defined for a networking environment, for this function call to be valid, the PEP must be

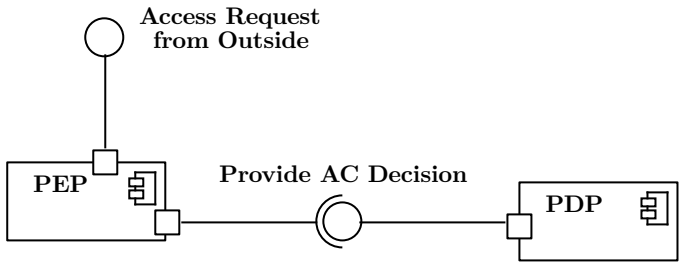


Figure 3.2: The components and interfaces of the DFlask architecture.

able to communicate with the PDP – in a way where the integrity and authenticity of the communication is ensured – as a precondition. As postcondition the PEP enforces the decision of the PDP, which means that the requested operation will only be executed if the PDP has decided that this is allowed.

The Policy Decision Point. The functionality of the PDP is – independent of whether it is a local or network-located PDP – that it has to decide on the access request of the PEP. This can be modelled as a function call initiated from the PEP. The input parameter of this function is thereby the access request. The output of this function is the decision calculated on the basis of the security policy. The precondition therefore is, that an AC policy is installed at the PDP.

3.2.3 The Extended Flask Architecture

The ExFlask architecture presented in section 2.1.4 consists of five elements: one PAP, one PIP, one PDP, one Policy Repository and a set of PEPs. Its components and their interfaces can be modelled as depicted in figure 3.3.

The Policy Administration Point. The PAP is responsible for modifying the AC policies stored in the Policy Repository. This functionality can be modelled as a function call from outside the architecture where the input parameters are the subject that wants to modify the stored policies and the changes that should be applied. Because of allowing arbitrary subjects to modify the policies in arbitrary ways would counteract the goal of defining and enforcing AC policies, this thesis assumes that the precondition of this function is that the requested change of the stored policies must be allowed by the policies stored before. The function then provides a success indicator, whether the operation was successful executed or not as output. As postcondition the policies stored in the policy repository are modified.

The Policy Repository. The policy repository itself has only the functionality of storing the policies that should be enforced. With this the policy repository is involved in every policy changing operation from the PAP. This can be modelled as a function call from the PAP where the input parameters are the change request (signalling whether to change an existing or adding a new policy as well as a

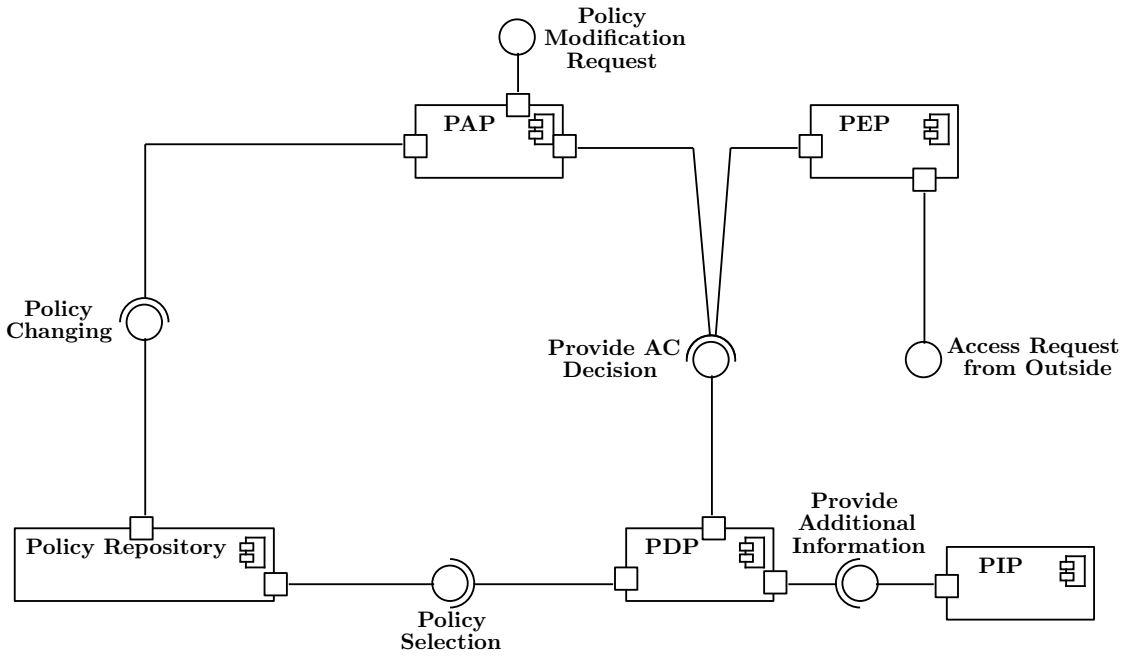


Figure 3.3: The components and interfaces of the ExFlask architecture.

description of the changes to apply) and the output is an indicator on success or failure. As precondition, it must be ensured that the PAP is the caller of this function. Because with this the PAP is the only component that can call this function and the PAP is already checking if the requested modification is allowed, this check can be omitted here. The postcondition of this function is then, that the change of the stored policies was done as requested.

The Policy Decision Point. The component responsible for deciding whether an access request should be allowed or denied is the PDP. Its functionality of deciding on access requests depends on the policies stored in the policy repository and receiving additional information from the PIP. When modelling the decision of the PDP as a function called from a PEP, the preconditions are that the function is actually called from a PEP and the PIP can be requested to give additional information. Thereby the input of this function is the access request from the PEP and the output is an indicator whether the access request has to be allowed or denied.

The Policy Enforcement Points. When a subject tries to access a controlled resource, this access is intercepted by the PEP responsible for the requested object. This can be modelled as a function called by the subject. Thereby the function gets the access request as input and provides the decision of the PDP as output. With this the precondition of this function is, that the PDP is available. Because of the functionality of the PEP to enforce the decisions of the PDP, the postcondition of this function is that the operation specified in the access request will be performed if allowed.

The Policy Information Point. For deciding on access requests the PDP can request additional information from the PIP. This can be modelled as a function called from the PDP which gets an identifier for an entity and a specification on which information were needed as input and outputs the requested information. With this the only precondition is that the function was called from the PDP.

3.2.4 The Architecture for the eXtended Access Control Markup Language

The XACML architecture presented in section 2.1.5 consists of seven elements: a PAP, a PRP, a PDP, a PIP, a set of PEPs and RAPs and a context handler. Its components and their interfaces can thereby be modelled as depicted in figure 3.4.

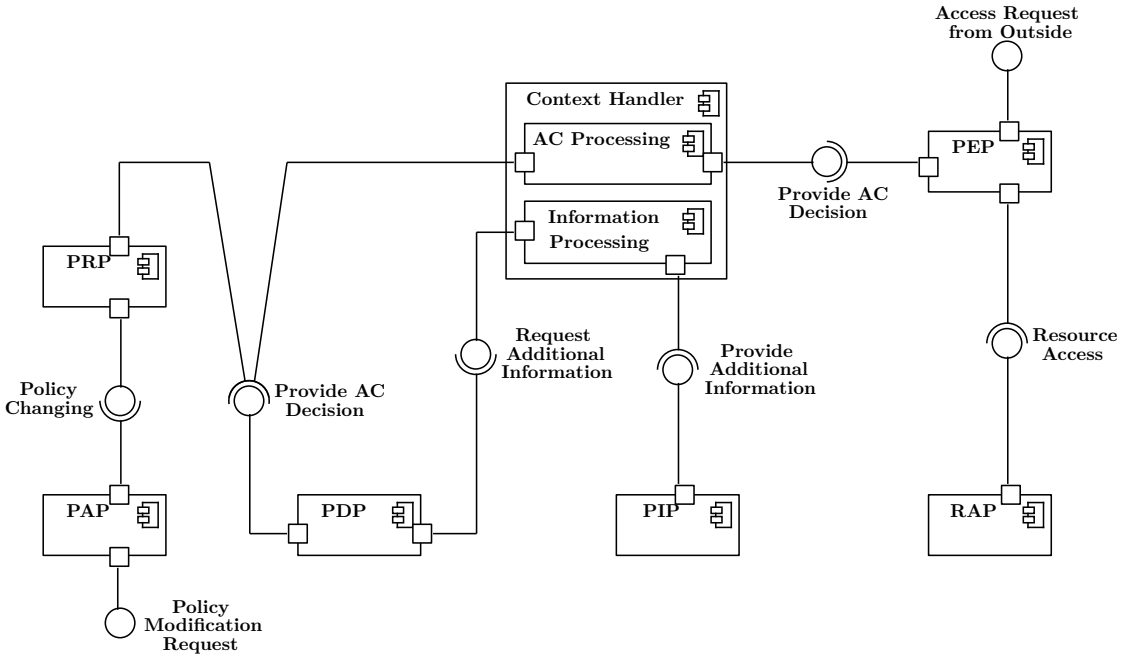


Figure 3.4: The components and interfaces of the architecture for the eXtended Access Control Markup Language.

The Policy Administration Point. The PAP is responsible for modifying the AC policies stored in the PRP. This functionality can be modelled as a function call from outside the architecture, where the input parameters are the subject that wants to modify the stored policies and the changes that should be applied. Because of allowing arbitrary subjects to modify the policies in arbitrary ways would counteract the goal of defining and enforcing AC policies, this thesis assumes that the precondition of this function is, that the requested change of the stored policies as an access request is allowed by the policies stored before. This function then provides an indicator, whether the operation was successful executed or not as output. As postcondition the policies stored in the policy repository are modified.

The Policy Retrieval Point. The PRP has the functionality of storing the policies that should be enforced. With this the PRP is involved in every policy changing operation from the PAP. This can be modelled as a function call from the PAP where the input parameter is a change request (signalling whether to change an existing or adding a new policy as well as a description of the changes to apply) and the output is an indicator on success or failure. As precondition, it must be ensured that the PAP is the caller of this function. Because with this the PAP is the only component that can call this function and the PAP is already checking if the requested modification is allowed, this check can be omitted here. The postcondition of this function is then, that the change of the stored policies was done as requested.

The Policy Enforcement Points. The PEPs are responsible for intercepting access requests and enforcing the decisions of the PDP. The functionality of a PEP can therefore be modelled as a function called by a subject. This function gets the access request as input and provides an indicator, whether the execution was allowed or denied, as output. Since the XACML architecture defines obligations that are executed on the resources in conjunction with an allowed access request, the postconditions of this function is the enforcement of the decision of the PDP as well as the execution of the specified obligations in the case of an allowed access request.

The Resource Access Points. In the XACML architecture for the PEPs obligations can be specified that are executed in conjunction with the enforcement of an access request. These obligations are actions that are performed on the controlled objects, which are accessible through a RAP. This behaviour can be modelled as a function called from a PEP. The input of this function is thereby the object that should be accessed, the operation of this access and the parameters of the operation. To let this function call be valid in the XACML architecture, it must be ensured in terms of a precondition that the PEP performing this access has received an allowing decision from the PDP for this access. The output of this function is then an indicator whether the execution was successful or not.

The Policy Decision Point. The PDP is responsible for deciding whether an access request should be allowed or denied according to the stored policies. For this it gets the access request from the context hander in the XACML format. Because of this, the functionality of the PDP can be modelled as a function called from the context handler which gets the access request as input and outputs an indicator whether the request is allowed to the stored policies. The preconditions of this function are thereby that the function is called by the context handler and the access request is given in the XACML format.

The Policy Information Point. The PIP is responsible for providing additional attributes to the PDP upon request. Because of the indirection in communication it does not get the request directly from the PDP but from the context handler. This behaviour can be modelled as a function called from the context handler, that gets

the identifier of the entity the attributes are requested for as input. Because of the PIP getting its information from the environment, the precondition of this function is – next to the ensuring that the request was sent from the context handler – that the environment can process the request for attributes. This function provides the set of attributes received from the environment as output.

The Context Handler. The context handler of the XACML architecture is responsible for four communications. At first, it is responsible for translating an access request, given from a PEP in its native format, into the XACML format and sending it forward to the PDP. At second, it is responsible for sending the decision of the PDP, received in the XACML format, back to the PEP in the PEPs native format. At third, the context handler is responsible for passing the request for attributes from the PDP to the PIP. At last, it is responsible for sending the attributes received from the PIP back to the PDP.

The context handler's functionality of sending an access request from a PEP to the PDP can be modelled as function called from a PEP. This function gets the access request as input and provides the decision of the PDP as output. For this it must be ensured in terms of a precondition, that the request was sent by a PEP and is in the native format of the PEP. In terms of a postcondition it must be ensured that the decision of the PDP is returned also in the native format of the PEP.

The functionality of the context handler to send a request for attributes from the PDP to the PIP and sending the attributes back to the PDP can be modelled as a function called by the PDP. This function gets the identifier of the entity the attributes are requested for as input. It then provides the requested attributes as output. Since this is only an intermediate step in the PDP there are no meaningful pre- and postconditions here.

3.2.5 The Policy Machine Architecture

The PM architecture presented in section 2.1.6 consists of six elements: an PDP, an PAP, an PIP, an EPP and a set of PEPs and RAPs. Its components and their interfaces can be modelled as depicted in figure 3.5.

The Policy Enforcement Points. The PEPs are responsible for intercepting access requests and enforce the decisions of the PDP on these requests. This can be modelled as a function called by a subject where the input is an access request and the output the decision of the PDP. Since this is the first interaction with this AC enforcement mechanism, there is no meaningful precondition. However, the postcondition of this function call is, that the EPP will be informed about this access request and the specified operation will be executed if this was allowed by the PDP.

The Resource Access Points. Accessing an object is done via the RAP responsible for that object. The RAPs are used from the PEPs to execute the operations of the access requests, when the specific requests were allowed by the PDP. This can be modelled as a function called from the respective PEP, where the input

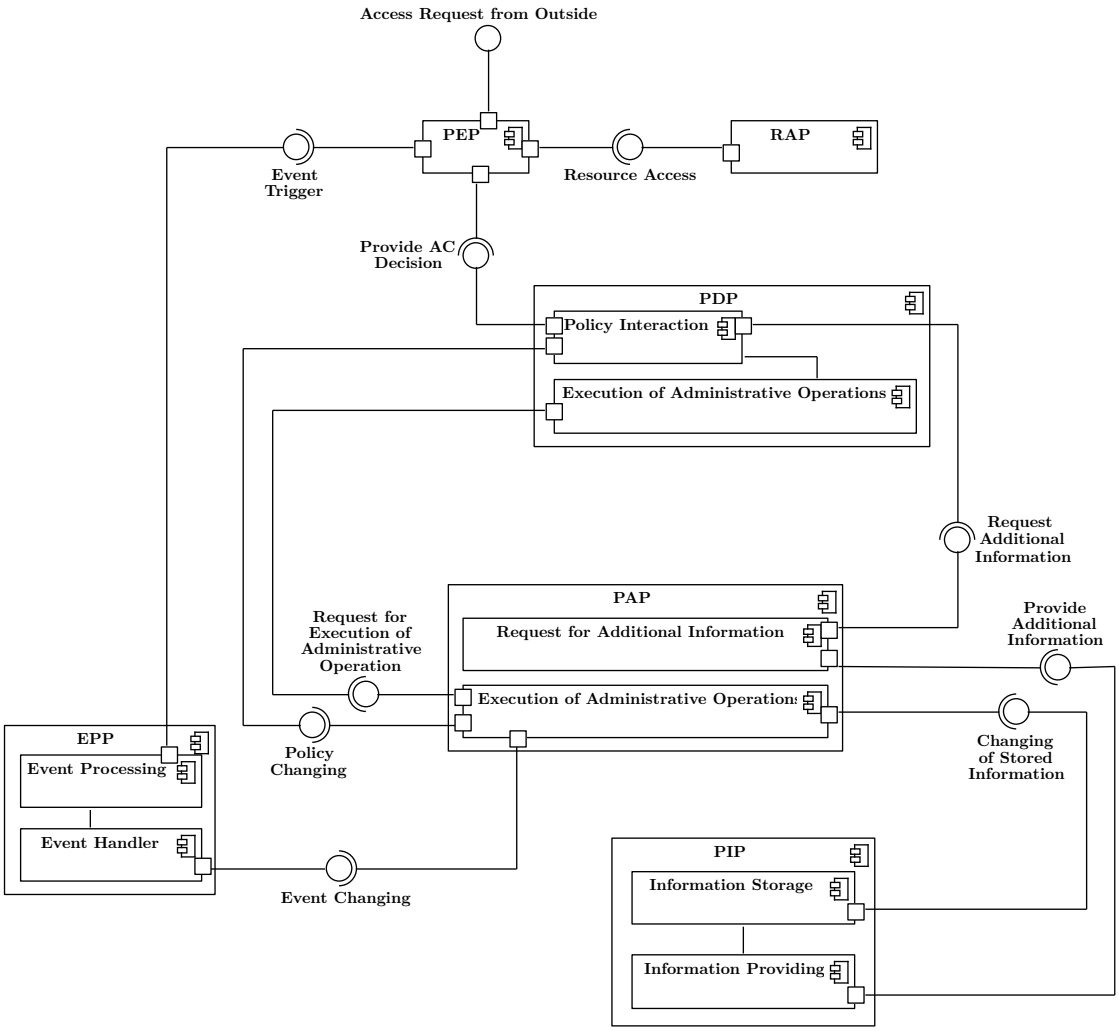


Figure 3.5: The components and interfaces of the Policy Machine/Next Generation Access Control architecture

parameters are the operation, the object and the parameters of the operation. The output of this function is then an indicator whether the operation was successfully executed. Since the RAP is only responsible for non-administrative operations the precondition of this function is, that the operation is a read or write operation. The postcondition is then, that the specified operation was executed and therefore that the postcondition of the operation holds.

The Event Processing Point. The EPP is responsible to react upon events that are triggered by the PEPs when an access request was allowed. This can be modelled as function at the EPP called by a PEP, where the input is the access request. Because of executing actions on behalf of an event is asynchronous, there is no meaningful output here. However, before any action on behalf of an event is done, the precondition of the access request being allowed must be fulfilled. The postcondition is then that the policy is altered according to the defined event handlers. This is done by utilising the PAP and can occur in terms of changes in the configuration of the PIP or the EPP itself.

The second functionality of the EPP is that its internal state can be modified during an administrative operation. This can be modelled as a function called by the PAP where the input is a specification of the altering and the output an indicator whether this altering was successful. For this call to be valid, the precondition of the requested altering is allowed by the enforced AC policy must be fulfilled. After the successful execution, the postcondition that the EPPs internal state has changed holds.

The Policy Administration Point. In the PM architecture the PAP is responsible for executing administrative operations. Because of the PAP is not reachable from outside, all requests for administrative operations are first received from a PEP which sends them to the PDP. The PDP then sends the operation request to the PAP if the operation is allowed by the AC policy. With this the execution of administrative operations can be modelled as function called by the PDP where the input is the operation to execute (including their parameters) and the output a success indicator for the execution of this function. The precondition of this function is that the operation is an administrative operation. After a successful execution the postcondition holds, that the PAP has executed the operations changing the state of the enforcing mechanism. This leads to changes in the EPP, PIP or the PDP.

Since the PM architecture does not provide any direct communication channel between the PDP and the PIP, the PAP is also responsible for forwarding attribute requests from the PDP to the PIP. This can be modelled as a function called by the PDP, where the input is the entity for which additional information are requested and the output the information the PIP stored for this entity. The only meaningful precondition for this function is that the PDP is requesting the additional information because they are required for deciding on an access request.

The Policy Information Point. The PIP is responsible for providing additional information to the PDP required to decide on an access request. Because of the indirection in the communication between PDP and PIP, this can be modelled as a function called by the PAP were the input is the entity for which additional information are required and the output the information the PIP stored for this. Since it is already enforced by the PAP that the PDP only calls this function when it has to decide on an access request, there is no meaningful precondition here.

The second interaction the PIP has to deal with is a request for changing its internal information according to the processing of an event handler or an administrative operation. This can be modelled as a function called by the PAP where the input parameters are specifying the alterations of the internal state of the PIP. The output of this function is then an indicator whether the altering was successful or not. Because every request of this type goes at least through the PAP – with which it is enforced that the altering is allowed according to the AC policy – there is no meaningful precondition here. However, after a successful execution of the altering request the postcondition holds, that the internal state of the PIP has changed.

The Policy Decision Point. The PDP is responsible for deciding whether an access request should be allowed or denied according to the AC policy. Thereby it must be differentiated between a normal access request on an object or an access request for an administrative operation. Both deciding functionalities can be modelled as function called by the PEP. However, in both cases the input is an access request.

For an operation on an object there is the precondition, that the operation specified in the access request is a read or write operation on an object. The output of this function is then the decision of the PDP based on the AC policy.

In contrast, for deciding on an administrative operation there is the precondition, that the operation specified in the access request is an administrative operation. Because of the PDP sends an allowed administrative operation to the PAP which executes them, the output of this function is an indicator combining whether the operation is allowed and if it was successfully executed. With this the postcondition of the PDP sending the allowed operation forward to the PAP has to hold.

Since with executing an administrative operation also the AC policy as internal state of the PDP may be modified, the PDP also has to handle request on changing its internal state. This can be modelled as a function called by the PAP where the input is the new policy to enforce and the output an indicator whether the change of the policy was successful. The precondition for this is, that the requested change is allowed by the AC policy enforced before the change. After a successful execution, the postcondition of the new policy being enforced holds.

3.2.6 The AppSPEAR Architecture

The AppSPEAR architecture presented in section 2.1.7 consists of three elements: a TEP, a TPS, and a set of TOMs. These components and their interfaces can be modelled as depicted in figure 3.6.

The Trusted Object Managers. The TOMs are responsible for intercepting access requests and enforcing the decisions of the TPS on them. This can be modelled as a function called by a subject where the input is the access request and the output is an indicator whether the access request is allowed or not. After a successful execution of this function, the postcondition of the operation being executed (if allowed) holds.

The Trusted Policy Server. The TPS is responsible for deciding on access requests given from a TOM. This can be modelled as a function called by a TOM. The input of this function is thereby the access request on which the TPS should decide. The output of this function is then an indicator whether the request is allowed according to the AC policy. For this in AppSPEAR exists no meaningful preconditions. However, the postcondition of this function is that if the request was allowed, an event for the TEP is triggered.

The Trusted Event Processor. The TEP is responsible for two functionalities. At first, it is responsible for reacting on the events triggered by the TPS. This can

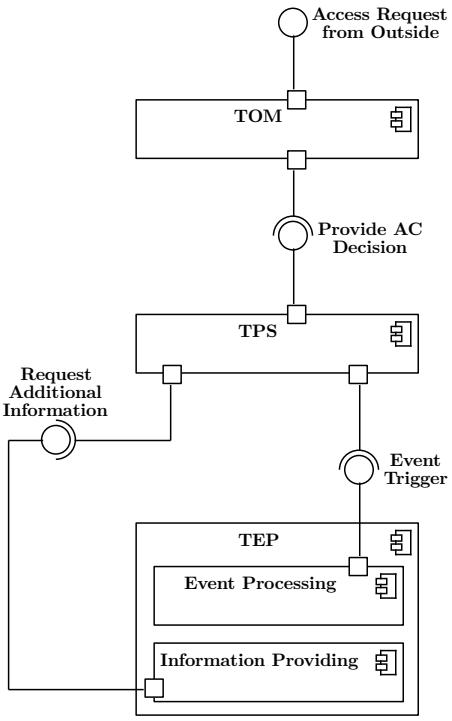


Figure 3.6: The components and interfaces of the AppSPEAR architecture.

be modelled as a function called by the TPS where the input is the access request. Because of triggering an event is asynchronous, there is no meaningful output here. The precondition of this function is thereby, that the access request was allowed by the TPS. After a successful execution of this function the postcondition holds, that the state of the policy is altered according to the defined event handler.

The second responsibility of the TEP is to look up additional information on behalf of a request from the TPS. This can be modelled as a function called from the TPS where the input is the entity for which additional information should be provided, and the output are the information known to the TEP. With this there are no meaningful pre- and postconditions here.

3.2.7 Comparison of the Access Control Enforcing Mechanisms

This section is to summarise the commonalities and differences of the analysed AC enforcement mechanisms. All the analysed architectures have in common that they define at least one PEP and exactly one PDP. In addition, most of the architectures are also defining a PAP and a PIP. Also, two more recent architectures are also defining an EPP. For a detailed comparison see section A.2.

The Policy Decision Point. In all architectures the PDP is the component responsible for deciding if an access request should be allowed, according to at least one installed AC policy. In all the presented architectures it therefore gets an access request from an PEP as input and returns an indicator on allow of this

request as output. Despite this core functionality there are differences on how the policies are stored and what additional functionalities a PDP can have.

The Policy Enforcement Point(s). A PEP is always responsible for intercepting access requests, sending them to the PDP and enforcing the decision of the PDP. This can always be modelled as a function call from a subject where the input is an access request and the output an indicator whether this access is allowed by the AC policy. However, there are differences between the architectures on how many PEPs are defined and what additional functionalities a PEP have. For instance the DFlask architecture just defines one PEP per network node, while in the other architectures as many PEPs as object managers in the system exists. Also, differences can be found on how the communication with the PDP is done. In the most architectures the PEPs communicate in a direct manner with the PDP. However, in the XACML architecture this communication is done via the context handler.

The Policy Administration Point. Most of the presented architectures define a PAP. This is a component always responsible for changing the set of enforced policies. For this it gets a change request (a special access request) as input and returns an indicator on success of the change as output. In addition, for this operation it is checked (directly by the PAP or indirectly by another component), if the requested change is allowed by the currently installed AC policy. However, there are differences on where the PAP is placed, what can be additionally changed by the PAP and what additional functionalities this component may have between the architectures.

The Policy Information Point. Most of the presented architectures are also defining an PIP. This component is always responsible for giving additional knowledge to the decision function of the PDP. For this the PIP gets the entity for which additional information are required as input and returns the information it stores about that entity as output. However, there are differences between the PIPs of the architectures in terms of the component that triggers a request to the PIP. Additionally, there is a significant difference between the PIPs of the architectures in terms of whether the information they contain can be modified.

The Event Processing Point. The PM and the AppSPEAR architecture are also defining an EPP as a component that executes specific actions on behalf of an allowed access request. For this it always gets an access request as input and returns no output because of asynchrony. However, before any action is executed, the precondition of the access request being allowed must hold. If this is true and the execution was successful, the postcondition of an altered policy state holds. The difference between the EPPs of the architectures is that the internal state of the EPP can be modified in the PM architecture but not in the AppSPEAR architecture.

3.3 Requirements Analysis of Access Control Policies

One goal of this thesis is to develop the functional requirements of state-of-the-art AC policies. For doing so, this thesis focuses on AC policies that are developed according to a state-of-the-art AC model calculus with a very high expressive power. Such a calculus is the DABAC calculus [SA23]. For developing the functional requirements of such polices this thesis reviews what functionalities are necessary to enforce an DABAC policy and how these requirements can be mapped to and influence the requirements summarised in section 3.2.7.

Basic Model Elements. An DABAC model consists of one or more sets of entities, one or more sets of attributes which can be assigned to the entity sets and a set of permissions. For all of these components hold, that they must be stored in a manner where the property of not allowing unauthorised manipulation holds.

To allow for expressing that an entity has specific attributes, a mapping from the entity to one or more attributes exists. Thereby each of these attributes can be a single attribute value (direct attribute assignment) or again a mapping to one or more attributes (indirect attribute assignment). These mappings must also be stored in a way where their integrity and consistency is ensured. The integrity and consistency additionally must also be ensured by communicating the entities in an environment which can not be assumed to be free of faults or attackers.

The storage of this information can be done in the abstractions of the target domain of an OS. However, it may not be possible to store all information in this way. Also, storing the information, required for the enforcement of an AC policy, distributed in an OS may not be as performant as possible when a lookup is performed. With this there is the requirement of having a PIP as component where an efficient lookup and storage of the entities and their attributes is possible.

AC Rules. Expressing the AC rules is done in terms of *Authorisation Predicates*. They consist of a set of preconditions that must be fulfilled and an – possibly empty – set of postconditions, that are ensured by changing the internal model state. As for the model entities, these predicates must be stored in a way where they are protected from unauthorised manipulation. Additionally, they must be stored in a way where they can be interpreted and executed in terms of checking their preconditions and ensuring their postconditions (if any). The storage and protection of the authorisation predicates can – according to section 3.2.7 – done at the point where they are evaluated or at a separate component only responsible for storing the authorisation predicates.

The evaluation of these predicates is done by the PDP. Because this is an action that does not need to make any changes on its data set, it must be ensured that every access of the PDP to the authorisation predicates can not modify these. This can be modelled as a function call from the PDP to the storage of the authorisation predicates, where the postcondition of the operation was read only has to be ensured.

Dynamic Policy State. To enable changes of the internal model state, an DABAC model is a deterministic automaton. The state space of this automaton is any possible combination of entities and attributes. This means that the sets of operations and authorisation predicates are fixed. The only model elements that can be changed are the entities and their attributes. With this the enforcement mechanism must allow changes on the entities, attributes and the assignment of attributes to entities. The changing of these model elements is done via *administrative operations*. These are special authorisation predicates whose postconditions describe how to change which model element.

With allowing components to undergo state changes, there is the additional requirement to ensure that these changes are done in a consistent manner while ensuring that these changes are authorised. Because of this the PIP needs a dedicated interface for operations which are performing changes on any of these model elements. This interface has to be the only possible way to change the state of such an element. A call to this interface should then only be valid if the change is allowed (the preconditions of the authorisation predicate are fulfilled) and the change is done in a consistent manner (for example ensuring that a string value can not be assigned to a boolean attribute). Since with such changes also new entities might be created, all entities must be distinguishable from each other.

Policy Administration. Despite changing model elements according to a fixed set of authorisation predicates, in a real world scenario there might be the necessity to perform operations that are considered as administrative but are not allowed by the DABAC model calculus. Such operations are the changing of the authorisation predicates or the attribute value domains. These operations can for instance be used to correct a wrong policy. However, such operations should not be possible just because a certain user of an OS is performing such an operation, but because the requested operation is allowed by the installed AC policy.

With this there exists two tiers of *administrative operations*. The first tier are the operations that are considered as administrative by the DABAC model calculus. The second tier is the changing of the authorisation predicates or other model components that are not defined as dynamic for DABAC models. However, operations that fit to the second tier of administrative operations are – from a policies point of view – operations to install a new and different policy for which a new verification has to be done.

CHAPTER 4

Architecture Design

For this thesis' goal to develop the functional and non-functional requirements of an RM architecture, in chapter 3 the requirements of the enforcement mechanisms presented in chapter 2 and the requirements of DABAC policies were analysed. Out of the intersection of the requirements of the presented mechanisms and the requirements of DABAC policies, this section develops an RM architecture that is designed to be implemented in an operating system and applicable for all kinds of policies that are analysed in section 3.3.

For doing so, at first – in section 4.1 – the design is developed in terms of the basic required components. This includes the modelling of specific communications in a form similar to UML sequence diagrams (see section A.1.2 for a legend). Then, in section 4.2 the design is refined to what interfaces these components provide and what is their input and output as well as their pre- and postconditions.

4.1 Basic Components

According to section 3.3, state-of-the-art AC policies require functionality to have knowledge about the controlled entities and additional information of them. Also, a mechanism to change the set of entities and the additional information of them as well is required. For enforcing a policy also a storage and evaluation of the policy itself is required. With evaluating the policy, a functionality to execute specific actions to alter the policy state is also required. According to section 3.2.7 an enforcement mechanism also contains a set of components to enforce decisions taken according to the policy.

Components of the Architecture. Taking into account these considerations, a mechanism for enforcing access control policies can be modelled as depicted in figure 4.1. It consists of a set of PEPs, one PDP, one PIP and one EPP.

To avoid increased complexity – which would counteract the goal of verifiability – there is only one component responsible for managing the entities and their additional information. The same holds for the component(s) responsible for storing and evaluating the policy. This has the additional advantage that all components know with which other component they have to communicate. Also, this could lead to a better system performance.

The Policy Enforcements Points. Thereby the components required for enforcing the policy are the PEPs. They must be implemented in a way where they can intercept any operation of the OS and enforcing the decision of the policy

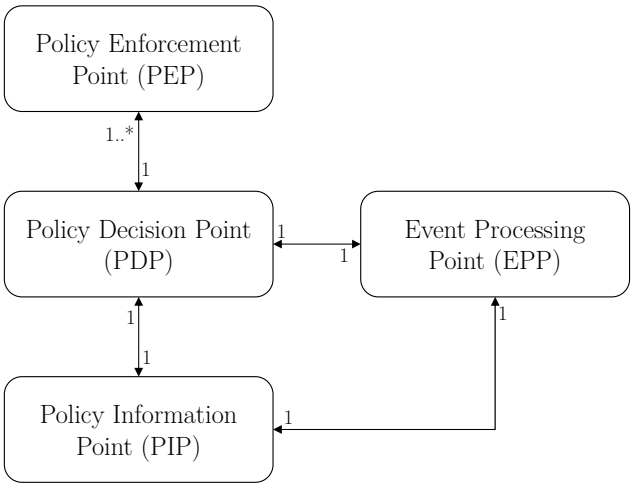


Figure 4.1: An overview of the components and their communication behaviour of the RM architecture. Directed edges model directed communication. Numbers denote the multiplicity of the components in the respective communication.

evaluation logic – which is located outside the PEPs – on that operations. For doing so they are included in the OS' object managers and responsible for intercepting any access request of them. For this a PEP sends an intercepted access request to the PDP and enforces the decision received from the PDP.

The Policy Decision Point. The PDP itself is then responsible to evaluate the policy. For this it must query the policy which must be stored somewhere. Since the PDP is the only point where the policy will be read, this thesis argues that the storage of the policy will be done at the PDP.

Number of Policies. In contrast to application-specific AC enforcement architectures presented in section 2.1 this thesis argues that there is no need to store and enforce multiple policies at the same time. In case multiple policies should be enforced at the same time, this leads to some form of policy merging and conflict resolution. This task could be done inside the enforcement mechanism. However, this would counteract the goal of letting the mechanism be small enough to be accessible for formal verification. With this, this thesis argues that the task of combining multiple policies to one enforced policy has to be done offline (outside the RM architecture).

The Policy Information Point. To enforce an DABAC policy the PDP must have knowledge about the attributes of the entities involved in an access request. Since these attributes depending on the policy to enforce, they can not be included in the metadata the OS holds for the controlled entities. Because of this they must be stored at some other point. For this section 3.2.7 defines the PIP is as a component which enriches an access request with additional information upon request.

The Event Processing Point. When enforcing a DABAC policy, there may be authorisation predicates that define postconditions on the policy state. To fulfil these postconditions, the PIP's stored information must be altered. However, the PIP should only be allowed to make such alterations if they were made according to an authorisation predicate whose preconditions are fulfilled. To ensure that this thesis' RM architecture is well-structured, the PDP should only evaluate the policy it stores and not modify their state. Because of this another component is required to performing the changes of the policy state. According to section 3.2.7 an EPP is responsible for executing specific actions on behalf of an allowed access request. Since this matches the definition of enforcing the postconditions of an DABAC policy, in this architecture the EPP will be responsible for that.

Enforcing a Policy. The process of enforcing the policy upon an access request is now as depicted in figure 4.2. When an PEP receives an access request it forwards it to the PDP, to let it decide whether the access is allowed or not. The PDP therefore requests additional information from the PIP and calculates its decision with the information given from the PEP and PIP. If the decision is to allow the requested access, the PDP sends it to the EPP. When the EPP receives a message that an access request was allowed by the PDP, it executes the specified post condition, which leads to changes in the PIP. After executing the specified changes, the EPP sends an indicator on success back to PDP to let it know that the execution has finished. After that – and independent of whether the decision allows or denies the requested access – the PDP sends its decision back to the PEP who enforces this decision.

Entity Modification. With using the described workflow, a PAP is not necessary in this design, because of the target domain of the AC enforcement mechanism to run in. With having an OS as runtime environment, all administrative operations of the policy are modifying elements controlled by the OS. This is because such operations are performed at the OS component responsible for handling the regarding element. However, all operations performed by such an OS component are intercepted by an PEP. Since the PDP is forwarding every allowed access request to the EPP, there is no difference at the PEPs regarding if the operation is an administrative operation or not.

Enable Policy Modification. However, this architecture does not define any component responsible for actions that can be considered as administrative on the second tier (policy modification). However, if such operations should be possible, an additional component – the PAP – could be defined to be responsible for modifying the stored policy and additional information required for that. With this the architecture could be modelled as depicted in figure 4.3.

Executing Policy Modifications. With this the workflow of executing an administrative operation is as depicted in figure 4.4. When the PAP receives a change request, it first queries the PDP to check whether the request is allowed according to the currently installed policy. This includes all the steps as described

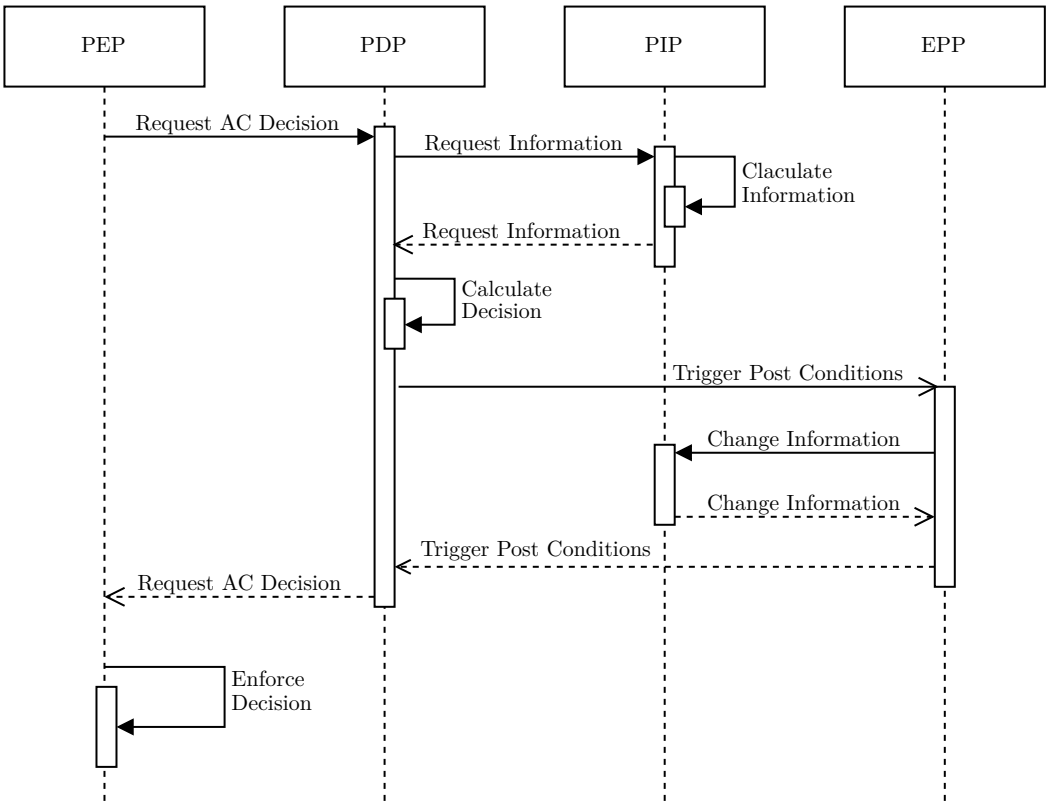


Figure 4.2: The communications for an access request.

for a normal access request (omitted in figure 4.4). If the request is allowed the PAP triggers the PDP, PIP and EPP to install the new policy and the information necessary for enforcing it.

4.2 Refinement of Components and Interfaces

The first architecture developed in this thesis consists of four elements: a set of PEPs, one PDP, one PIP and one EPP. These components and their interfaces can be modelled as depicted in figure 4.5.

The Policy Enforcement Points. The PEPs are responsible for intercepting access requests and enforcing the decisions of the PDP. This can be modelled as a function called by a subject. The input of this call is the access request and the output an indicator whether the access request was allowed. Because this is the entry point for interactions with the RM architecture, there are no meaningful pre- and postconditions here.

The Policy Decision Point. The PDP is responsible for deciding on access requests given from a PEP. This can be modelled as function called from a PEP. The input of this function is the access request on which the PDP should decide, and the output is an indicator whether the request is allowed according to the

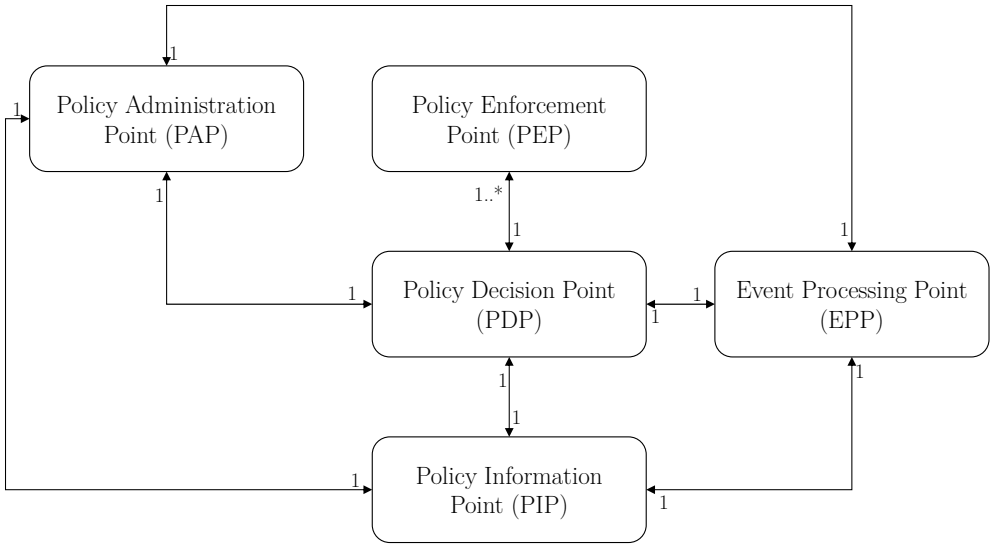


Figure 4.3: The high level design of the RM architecture including a PAP. Directed edges model directed communication. Numbers denote the multiplicity of the components in the respective communication.

AC policy. For these operations there exists no meaningful preconditions. The postcondition of this function is, that if the request was allowed, it is sent to the EPP, notifying it to execute the policy state modification operations (DABAC postconditions) of this request.

The Policy Information Point. The PIP is responsible for providing additional information for an entity to the PDP upon request. This can be modelled as a function called from the PDP. The input of this function call is the entity for which additional information should be provided. Since this is a simple request for information there exists no meaningful pre- and postconditions. This function's output are the requested information.

Information Changing in the PIP. The PIP is also responsible for handling the information change requests triggered by the EPP. This can be modelled as a function called by the EPP. The input of this function are the changes to perform. Because this function is called by the EPP and the EPP is only performing a change request to the PIP if an allowed access request requires that, there are no meaningful preconditions here. The output of this function is an indicator whether the requested changes are successfully performed or not. After a successful execution, the postcondition of the internal state of the PIP is modified holds.

The Event Processing Point. The EPP is responsible to change the state of the installed policy by modifying the information stored in the PIP upon an allowed access request. This can be modelled as a function called from the PDP where the input is the access request. Since the policy state must not be altered if the access request was denied, the precondition of this function is that the access request given as input was allowed by the PDP. To prevent the PDP from making a

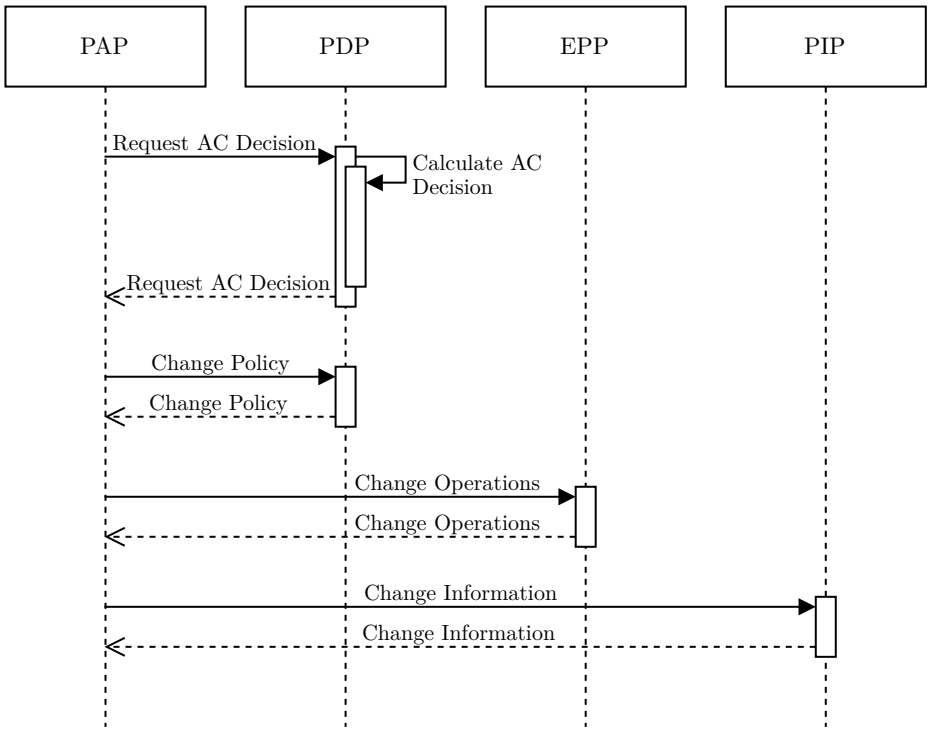


Figure 4.4: The communications for a change request.

decision based on outdated information, a new access request may only be processed after the rules specified in the EPP have been successfully executed. The EPP therefore outputs a success indicator after processing these rules. After a successful execution, the postcondition holds that the information stored in the PIP has been changed as specified.

Including a PAP. When including the PAP into this architecture as described above, this adds the PAP itself as component as well as interfaces to the EPP and PDP. Also, the interface of the PIP responsible for changing the stored information and the interface of the PDP responsible for providing decisions on access requests changes. With this the components and interfaces of this architecture can be modelled as depicted in figure 4.6.

The Policy Administration Point. The PAP itself is a special PEP responsible for administrative operations on the second tier. This can be modelled as a function called from a subject where the input is the change request to perform. The output is a combined indicator on allow and success of performing the changes. Since this a call from outside the architecture, there are no meaningful pre- and postconditions here.

Modification of the EPP. With adding the PAP, the EPP is now also responsible for reacting on requests of changing its stored operations. This can be modelled as function called by the PAP where the input values are the new operations to store. Before any modification of the internal state of the EPP is executed

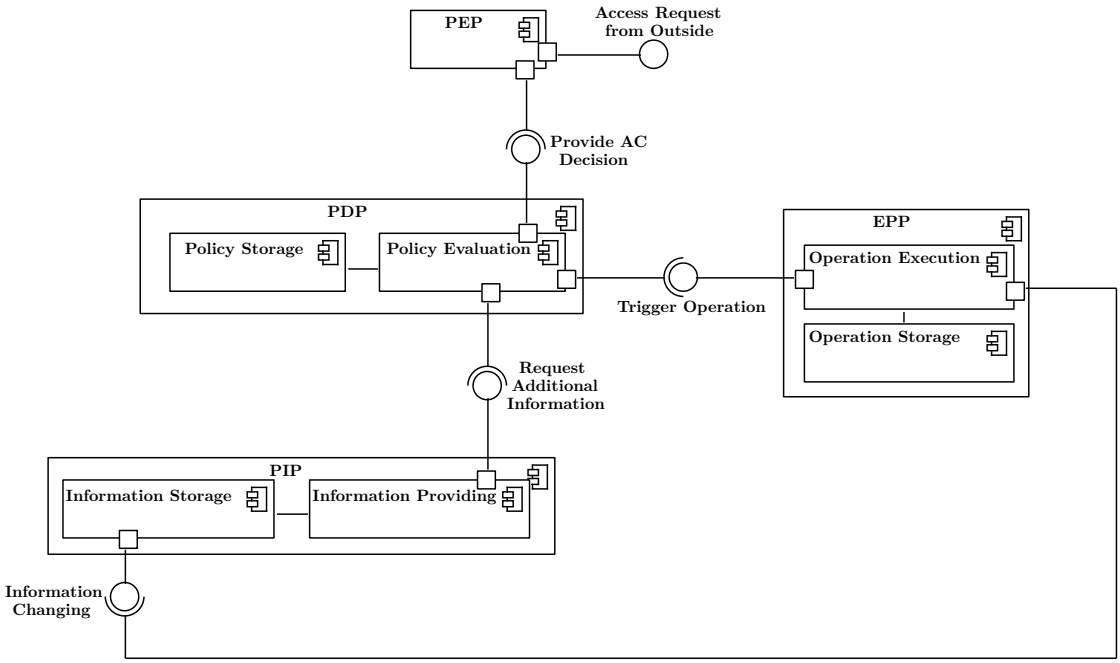


Figure 4.5: The components and interfaces of the first design of an RM architecture.

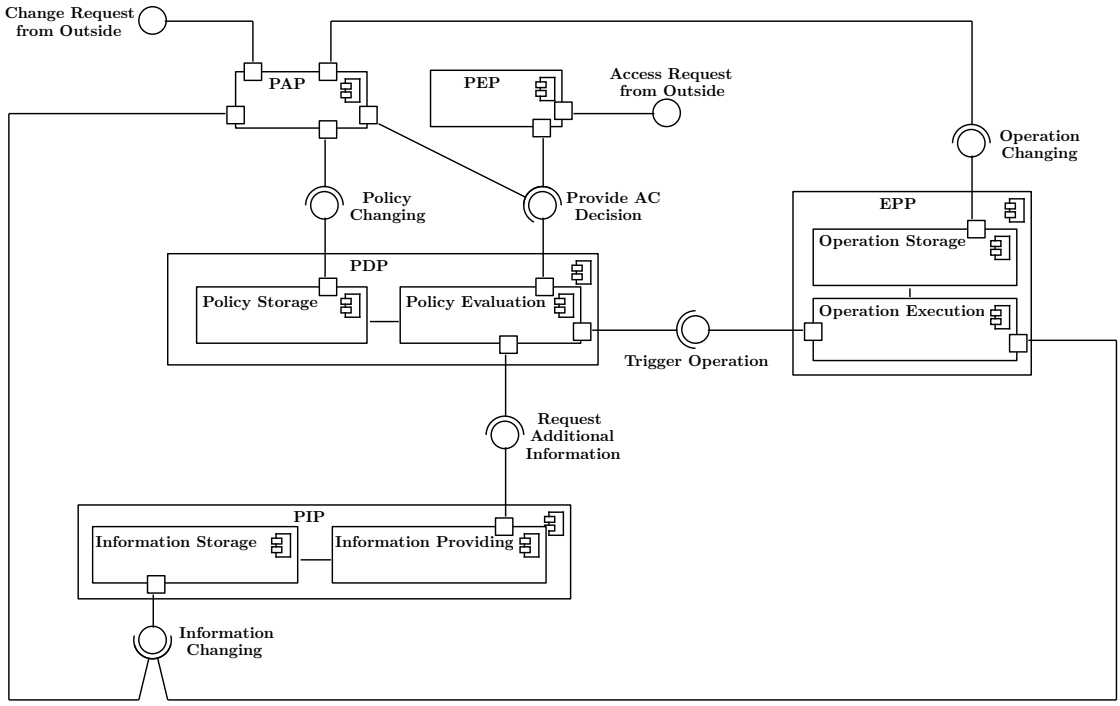


Figure 4.6: The components and interfaces of this thesis RM architecture including a PAP.

it must be ensured that these changes are allowed by the policy. This must be ensured in terms of a precondition. After a successful execution of this function the postcondition holds that the operations and the rules, which operations to execute for which type of access request, stored in the EPP have changed. The output of this function is then an indicator whether the change was successfully performed or not.

Modification of the PDP. With adding the PAP also the PDP gets another responsibility. This is for changing the policy stored and can be modelled as a function called from the PAP. As for the changes on the EPP, the precondition of these changes are allowed by the policy has to hold. After a successful execution of this function, the postcondition of the stored policy has changed has to hold.

Also, the interface of the PDP to provide decisions on access requests has to change. This function can now be called by the PAP or a PEP. For both of them there are no pre- and postconditions. Also, the input and output does not change.

Modification of the PIP. By adding the PAP, the function of the PIP to handle requests for changing the stored information is not only used by the EPP any longer: now also the PAP is using this interface. While this changes nothing on the input and output of this function, for a call from the PAP the precondition of the requested changes are allowed by the policy has to hold. The postcondition thereby does not change.

CHAPTER 5

Architecture Implementation

Based on the design developed in chapter 4 a reference implementation of an RM in one of the OS presented in section 2.3 should be developed to evaluate the practical feasibility of this design. For this in section 5.1 the possible alternatives of how to do this were discussed. Following this discussion, one of the alternatives will be selected for implementation. The essential details of the implementation will then be described in section 5.2. Based on these results, section 5.3 describes next research steps based on this implementation.

5.1 RM Implementation Alternatives

In this section alternatives on how to implement the developed RM architecture are discussed. This includes the OS into which the architecture is to be integrated (section 5.1.1), alternatives on how to integrate the RM into such an OS (section 5.1.2) as well as alternatives on how to store and load the data required for enforcing an AC policy (section 5.1.3).

5.1.1 Operating System Environment

In section 2.3 four operating systems were presented, from which one should serve as basis for the implementation of the RM architecture developed in the previous chapter. This section therefore gives an overview of the advantages and disadvantages of these operating systems.

The first OS presented in section 2.3 is Tock. Tock is designed for low power microcontrollers. Because of Tock is tailored to this domain, it is only useful for specific applications and is therefore not applicable for arbitrary policies. Furthermore, the memory management of Tock is more complex from an applications point of view than in other operating systems, since there is no illusion of infinite memory in Tock. Another disadvantage of Tock is the form of isolation between the capsules. Since the RM architecture have to be involved in every system operation, at least one of its components has to be implemented as a capsule. Because the isolation of the capsules is only done at compile time, an attacker could bypass it at runtime.

The second OS presented in section 2.3 is Theseus. Theseus is a reliable OS with focus on minimising state spill. With this it focuses on the enforcement of non-interference policies. However, this contradicts the goal of enforcing access control policies, because the behaviour of a PEP have to be influenced by the communication with the PDP.

The third OS presented in section 2.3 is Redox. Redox is a microkernel OS in an advanced stage of development. With this it runs most system services in userspace. Because of this it provides address space isolation even between these system services, which is a strong point in terms of robustness. However, because of its advanced stage of development, Redox is a very complex system. With this there exists many possible locations where a PEP has to be implemented. Here the two security features provided by Redox may help. However, the quality of the documentation is inadequate considering the complexity of Redox. In addition, there is no documentation provided on how to implement system services like the components of the RM architecture.

The fourth OS presented in section 2.3 is Poplar. Like Redox, Poplar is a microkernel OS but in an early stage of development. Despite using a microkernel, Poplar is a very simple system also at code level. With this it has clear defined interfaces and an easy-to-use structure. Because of this the implementation of the RM architecture becomes easier. Also, the security features already provided by Poplar can be used as entry point for the implementation, because the enforcement of this mechanism is done at the system call handler in the kernel. However – due to the early stage of development – Poplar cannot be assumed to be stable. For instance implementing additional things into the user space standard library can lead to a kernel page fault and things running on an RISC-V platform may not be able to run at an x86 platform.

5.1.2 Operating System Integration

In this section the possible ways on how to integrate the RM architecture into an OS are discussed based on three alternatives. The first one is the complete integration into the operating systems' userspace (section 5.1.2.1), the second the integration of all components in the operating systems' userspace except the PEP (section 5.1.2.2) and the third one the complete integration into the operating systems' kernel (section 5.1.2.3). Thereby for each alternative the advantages and disadvantages are discussed.

5.1.2.1 Complete Userspace Integration

Integrating the RM architecture completely in userspace means to implement the PAP, the PDP, the PIP and the EPP as userspace service processes. With this all communication between these components have to be done via IPC. Integrating the architecture completely in userspace also means to integrate the PEPs in the standard library used by the userspace processes.

With doing so there is the advantage that the RM components are isolated as strong as possible from the rest of the OS and from each other. Isolating the components from each other has the additional advantage, that an unauthorised manipulation of one of these components becomes hard, which is a point for the tamperproofness of the implementation.

On the other side, when implementing the PEPs in the standard library at userspace, not all operations can be controlled by an AC policy. In especial

the IPC mechanisms must not be controlled, because they are required by the communication between the RM components.

Also, with implementing the PEPs in the standard library there is no guarantee that the RM is involved in every possible system operation. Whenever a process is not using this library or doing a system call manually, the RM is bypassed.

5.1.2.2 Userspace Integration with Enforcement in Kernelspace

Integrating the RM in userspace with enforcement in the kernel means to implement the PAP, the PDP, the PIP and the EPP as userspace service processes but integrating the PEPs in the operating systems' kernel. With this again all communication between the PAP, the PDP, the PIP and the EPP has to done via IPC.

In contrast to a complete integration in the userspace, in this approach the PEPs are integrated in the system call handler routines in the kernel. Due to this all communication where a PEP is involved has to be done via shared memory.

Integrating the RM in this way has the advantage of a strong isolation between the OS and the RM and between the components of the RM due to address space isolation, which is a strong point in terms of tamperproofness. In contrast to the complete userspace integration, this approach has also the advantage that it can be achieved that the RM is involved in every system operation where AC can be used.

On the other hand this approach results in increased runtime costs for a system call. This is due to the necessary changes between kernel and user space for the enforcement of an AC decision. Initially, a system call is triggered by a process, which then switches to the kernel. At this point, the PEP requests the PDP to query whether this access should be allowed. As the PDP is running in userspace, another switch is made here. In order to make its decision, the PDP requests further information from the PIP, which causes an additional cost in terms of IPC runtime. The PDP then communicates its decision to the PEP, which triggers another switch to the kernel.

With this the OS, the architecture is integrated in, has to provide solutions for two problems. At first, it has to implement a mechanism to communicate values from the kernel to an userspace application, without the userspace application is explicitly requesting this.

At second the control flow itself leads to problems. While the first switch from the userspace into the kernel is as for a normal system call, the switches required for the communication between a PEP and the PDP may cause problems. Thereby the switch from the kernel into the userspace for the PEP requesting the PDP might be solvable through the invocation of the scheduler in combination with the preparation of the PDP.

However, the second switch from the userspace into the kernel for the PDP to communicate its decision is problematic, because of the semantics of a system call. To not violate the semantics of a system call from an applications point of view, the PDP has to switch to the same destination in the kernel code where it was called from, because of the PEP requires an answer from the PDP to proceed. This might be solved through a direct call to the scheduler from the userspace application of the PDP. Another solution approach might be, that the PDP does not return to

the same destination where it was called from. However, this would lead to an interrupted system call and the problem of how the return value of the system call could be given to the application.

5.1.2.3 Complete Kernelspace Integration

Integrating the RM architecture complete in the OS kernel means to implement the PAP, the PDP, the PIP and the EPP as Rust modules of the kernel and the PEPs in the system call handler in the kernel. With integrating everything in the kernel, now AC can be done over all system services useable for applications, and it can be ensured that the RM is involved in every such operation. Because with this approach everything is implemented in the kernel space it reduces the code the implementation depends on, which is a point in terms of the verifiability of the implementation.

On the other side this approach has a drawback in terms of isolation. Because everything is integrated in the kernel, there is no address space isolation between the kernel and the RM or between the RM components. However, isolation is achieved at compile time through the usage of Rusts visibility patterns, which provides a minimal form of isolation enforced by the compiler.

Another drawback of this approach is how the internal data structures of the components have to be implemented. While in the other approaches all necessary data structures can be implemented like in any other Rust application, in this approach they must be used with unsafe Rust because of the library like implementation of the components.

5.1.3 Internal Component Design

In this section alternative ways how the implement the data storages for the RM components are discussed. Thereby it is to note, that internal data storages are only required for the PDP, the PIP and the EPP because they have to either perform calculations or lookup information. Data storages are not needed by the PEPs because they only have to query the PDP and enforce its decisions. Also, the PAP does not need any kind of stored information because it only has to query the PDP and afterwards forward information to the PDP, the PIP and the EPP.

This section discusses four alternatives on how to implement the data storages of the mentioned RM components. The first approach is an in-memory approach (section 5.1.3.1). The second and third approaches are based on having non-volatile memory (section 5.1.3.2). Thereby the third approach extends the second approach by the possibility that the stored representations of the data can be read by humans. The fourth approach is for the PIP to fetch the requested attribute values from the environment (section 5.1.3.3).

5.1.3.1 In-Memory Representation

A simple way to implement a data storage is to hold all data in memory by using appropriate types. For the PIP this can be done by using a mapping from an

entities ID to the additional information stored for that entity. Thereby it must be assumed that all possible information can be represented within one type.

For the PDP having all data in memory, a list of functions can be used. Thereby each function has to accept the access request as parameter and returning a boolean value. The decision logic of the PDP then iterates over this list and calculates its decision from the results. However, requiring every function to collect the additional information needed on their own, would have a negative impact on the performance. Because of this the PDP collects this information before iterating over the functions. Then these values are passed as parameters to the functions stored in the list, requiring these to accept two additional parameters, one for each entity participating in the access request.

For the EPP having all data in memory, the procedure is similar to the one for the PDP. The EPP also can be defined as a list of functions, each one accepting an access request as input. The event processing logic of the EPP is then iterating over these functions. Thereby each function checks if it should execute upon the given access request. If a function determines that it should execute, it proceeds and executes its defined operations on the PIP, otherwise it terminates. The EPP collects all success indicators of these functions and builds another success indicator out of it which is sent to the PDP.

However, letting each function execute tier actions on its own, requires a public interface to the PIP. With this there is no guarantee that state modification on the PIP is only done by the EPP, which is a violation of the developed design because of violating the principles of isolation and tamperproofness. To solve this problems while using this approach, the behaviour of the stored functions could be changed. Instead of executing the specified actions on their own, each function returns a list of functions, it wants to execute, to the EPP. The EPP collects this functions and then executes them itself on the PIP. With this the PIP does not need a public interface any longer.

While solving the problems of isolation and tamperproofness, the approach of collecting the actions before executing them could have a negative impact on the systems' performance, because the EPP now has to iterate over two lists. To solve this, the way of defining the actions to take could be changed. One approach could be to define a mapping between access requests and a list of functions that have to execute upon this access request. This has the drawback that all parts of the access request have to match before any function can be executed.

Because of this, another variant could be used. Thereby the functions defining the decision rules of the PDP are not only returning a boolean value, but also a list of function identifiers. These function identifiers are then sent to the EPP, where each ID is mapped to function. The EPP then executes all functions defined in that way. However, this would require some kind of shared state between the PDP and the EPP and is a violation of the requirements from section 4.2.

5.1.3.2 Static Representation Variants

One alternative of having a data storage is to read them out of a static – existing across restarts – resource and store every change in this resource. However, this requires the representation of such resources in the OS like with a file system.

Problems with static resources. When dealing with static resources multiple issues have to be solved. At first the representation of static resources requires hardware interactions and abstractions. At second, it is necessary to implement measures, to ensure that the resource is tamper-proof in terms of both reading and writing. This could be achieved by utilising cryptographic hash functions. However, it is also essential to guarantee that the hash value has not been manipulated without authorisation, which introduces the issue of the root of trust.

Increased complexity. Assuming that these problems had been solved, the complexity of the implementation would have increased. This could potentially lead to difficulties in ensuring the verifiability of the implementation.

Expanding to be readable by humans. In the case of static resources, it is possible that they may be available in a human-readable form, such as a text file. However, the process of interpreting and correctly writing such a representation would further increase the complexity of the implementation, which in turn would have a negative impact on verifiability.

5.1.3.3 PIP only: Fetching from external sources

Assuming that interfaces to external hardware are available in the OS, it could be possible for the PIP to request the requested information from other system components in contrast to storing the information. This would eliminate the need for a data storage in the PIP.

However, this procedure would increase the runtime costs, as all the required information would first have to be collected from the environment before they are sent to the PDP. Furthermore, alterations to the interfaces of other system components may be necessary.

In the case where data required for an access decision is requested from other components, it is essential to guarantee its correctness and to ensure that it is exchanged in a tamper-proof manner. This would result in an increase in the complexity of the implementation, which would consequently have a negative impact on verifiability. Additionally, due to this dependency, the assumption of complete isolation of the RM cannot be made.

5.2 Details of the RM Implementation

In section 4.2 the design of the RM architecture was developed. Based on the discussions in the previous section, this section provides the decisions in which OS the RM architecture was integrated (section 5.2.1), about how this integration was done (section 5.2.2) and how the internal data structures are designed (section 5.2.3). Based on these decisions it is discussed how policy flexibility is achieved for the used approach (section 5.2.4).

5.2.1 Operating System Selection

Based on the discussion of the advantages and disadvantages of specific OS as basis for an RM implementation in section 5.1.1, two of them are pointed out to be useable for that. These are the microkernel OS Redox and Poplar. Despite using a microkernel, both of them are providing security features that may serve as entry point for an RM implementation.

However, there are differences on the stage of development and the quality of documentation between these systems. While Redox is at an advanced stage of development and therefore a complex system, Poplar is at an early state of development and with this more simple to understand. Also, the quality of Poplars documentation is better than the one for Redox.

With this, Poplar was selected to serve as basis for the RM implementation. However, because of its early stage of development, the usage of Poplar has also some drawbacks like its unstable behaviour. Because one can deal with this drawbacks by selecting appropriate implementation alternatives and Poplar is under active development these drawbacks are negligible when choosing the OS.

5.2.2 Integration of the Reference Monitor

The best way of balancing the non-functional properties of the RM and the OS is to integrate the RM components in the user space while placing the enforcement in the kernel space as described in section 5.1.2.2. With this there is tamper-proofing ensured through a strong isolation, and the RM is involved in any possible system operation.

However, Poplar does not provide any mechanism to communicate values from the kernel to the user space without a user space application is explicitly requesting this. Also in the current version of Poplar there is no possibility to solve the described problem of the control flow.

The next approach would be a complete integration into Poplars user space as described in section 5.1.2.1. With this approach the problems of the first approach are not present. However, in this approach it can not be guaranteed that the RM is involved in every possible operation. Unfortunately, due to an unknown reason, Poplar runs into a kernel page fault when the PEPs are implemented in the standard library (when compiled in debug mode) or fails to boot (when compiled in release mode).

Because of the errors described, the last approach was taken: integrating the RM completely into the Poplar kernel as described in section 5.1.2.3. In this approach the RM is involved in every system operation and verification becomes easier due to a reduced complexity. However, this approach lacks of a strong isolation.

5.2.3 Internal Data Structures

The most used data structure in the RM implementation is the access request. An access request as in definition 3.2 is thereby implemented as a structured type which consists of the subjects name, a string to identify the object and an enumeration value to identify the operation.

5.2.3.1 Internals of the PIP and EPP

Based on the entities access request, the PIP collects additional information about these entities. Due to the selected way of integrating the RM into Poplar, the interface of the PIP used for providing additional information is implemented as a function that receives an access request as input and returns the additional information as an instance of another structured type.

However, in Poplar not all system calls are working directly on identifiable objects. Due to this, this functionality is only implemented to provide additional information for the involved subject.

The type of the value returned from the function of the PIP used for information lookup is a wrapper around a vector of attributes that a subject may have. To have all possible attributes encoded in one type and let the implementation be as small as possible, the attributes are implemented as values of an enumeration type but other ways – e.g. encoding them in a string – are also possible. This type also offers functions to add or remove a subject attribute or to check whether a subject has a certain attribute.

The second interface of the PIP – responsible for changing the stored information upon request – is implemented as a function that receives the change to perform as input. The possible changes are thereby encoded in an enumeration type in which each value takes one or more argument and specifies an operation to execute on the storage of information in the PIP. In the implementation there are values provided for adding a new subject to the list of known subjects, removing a subject from this list, adding an attribute to a subject or removing an attribute from a subject.

The EPP is implemented in the way of a list of functions where each function has to check whether it should execute or not. This basic variant was selected to fulfil the specified requirements and due to the problems the advanced approach introduces. Thereby for not requiring a public interface to the PIP the variant was chosen where the EPP collects all operations and executes them on its own.

5.2.3.2 Internals of the PAP

The functionality of the PAP responsible for changing the state of PDP, PIP and EPP is implemented as another system call. This system call gets the addresses where the new information to store in the PDP, PIP and EPP are located in the memory and tries to create variables of the correct type out of them. If this was successful for all three addresses, the actual PAP code is executed in the name of the current subject.

The PAP at first queries the current policy, if changing the information in PDP, PIP and EPP is allowed. If this is the case, the changes will be performed. This is done via the functions in the components that are implementing the interfaces defined for this purpose. Thereby changing the information stored in PDP and EPP means to overwrite them with the new values provided.

In contrast to the interfaces of the PDP and EPP, the interface of the PIP responsible for changing the stored information is restricted since it only accepts specific operations. With this it becomes hard for a user to exchange all information stored in the PIP like possible for the PDP. Because of this, the interface of the PIP

responsible for receiving requests to change the internal stored information has been split into two different interfaces. The first interface is responsible for performing changes triggered from the EPP as described above. The second interface is responsible for performing changes triggered by the PAP. For this it gets the new information to store in a format that allows to simply overwrite the old information with the new one. With this the architecture of the implementation differs from the architecture developed in section 4.2 as depicted in figure 5.1.

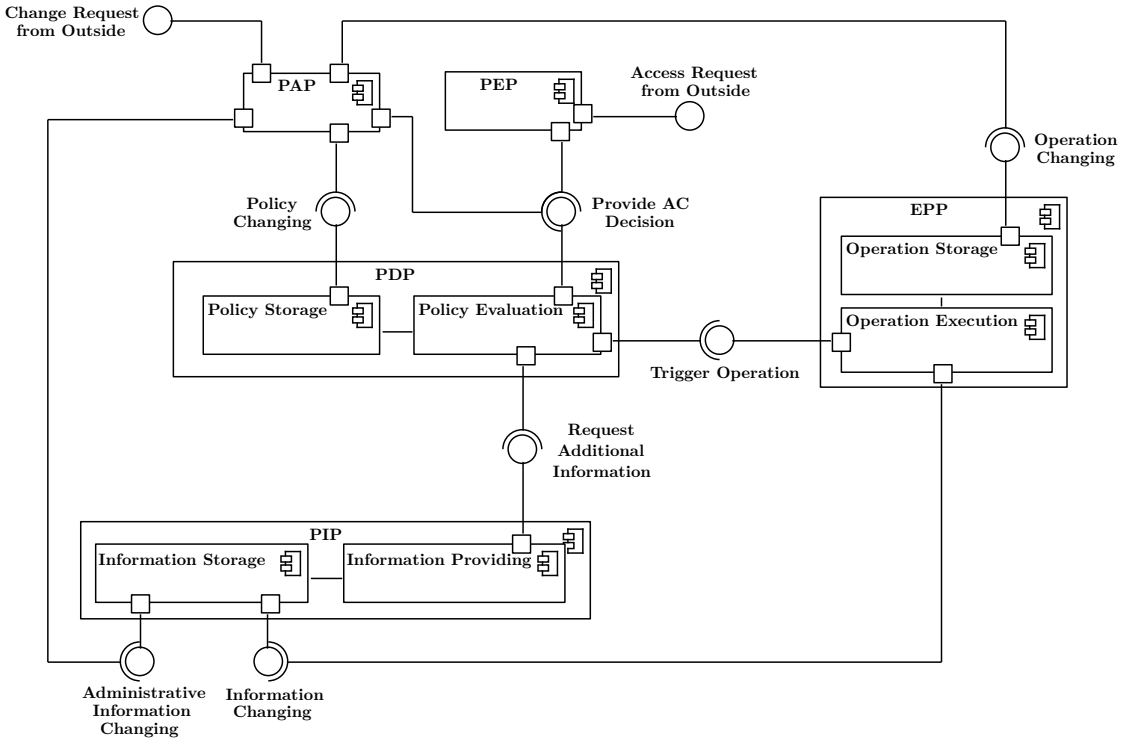


Figure 5.1: The design of the implemented RM architecture

5.2.3.3 Issues with writeable Storages

With implementing the functionality of allowing the information stored to be changed (triggered by the PAP or EPP), the variable containing the data must be allowed to be manipulated. Due to the implementation of the components of the RM architecture as modules in which certain functions needs to access a single storage, the storage of the policy rules in the PDP, the storage of the events in the EPP and the storage of the attributes in the PIP must be variables accessible from every function within this module. This is achieved by using a global variable.

However, the use of global variables is difficult in Rust, because it has to be ensured at compile time that the variable is initialised before any use as well as that every access to this variable is thread-safe and follows Rusts rules of ownership. In normal Rust programs or libraries all of these issues except the thread-safeness can be solved for the variables used in the RM implementation by choosing an appropriate type.

However, this is not possible for the Poplar kernel. Because it is not linked against Rusts standard library it can only use types that are defined by itself or its dependencies. With this a global variable that can be manipulated requires unsafe Rust.

Unsafe Rust is also required because of storing functions. This is because the type used for dealing with functions as variables does not implement specific markers which show to the compiler that values of this type can safely be shared between threads. Since with this it cannot be ensured at compile time that every access to the global storage of functions in the PDP or EPP is thread-safe, all operations on global variables dealing with this type must also be done via unsafe Rust.

5.2.4 Achieving policy flexibility

One goal of this thesis is to achieve policy flexibility in the developed RM architecture. Thereby policy flexibility means that a wide range of policies should be supported without changing fundamental parts of the architecture.

In the implementation of the RM architecture the enforced policy consists of a set of functions with the same signature. The exact signature of these functions can be changed as long as each function returns a boolean value and the signature is also changed for the system call given access to the PAP.

Because some policies requires additional information on behalf of an access request, the PDP queries the PIP. For this the PIP accepts an access request as input and returns a value of a structured type. The lookup itself is thereby done via a mapping from the identifier of the task given in the access request to the additional information known. The type of the additional information can be modified as long as the handling logic in the PDP and the system call giving access to the PAP are modified.

After evaluating an access request to be allowed, the PDP sends this access request to the EPP triggering one or more events. These events are stored as a list of functions each accepting an access request as input and having no output. Each of these functions can send arbitrary values from an enumeration type to the PIP, each value triggering a specific operation on the PIP.

Because of storing the policy rules and events in functions and storing additional information in a mapping, policy flexibility of the implementation is achieved. This is because the stored functions can do arbitrary computations and arbitrary information can be stored in the mapping. Also changing the types used for the values of the functions and additional information is possible with little effort with the design of the interfaces. With this the implementation follows the specification of the developed design and the design can be considered to be policy flexible.

5.3 Remaining Tasks

With the results of section 5.2 there are still some aspects related to the implementation of the architecture that have not yet been addressed. These are a performance analysis, a better isolation of the implementation and reducing the code dependencies in combination with a formal verification of the implementation.

CHAPTER 6
Summary

Due to the large number of critical applications relying on a secure operating system, unauthorised access to the controlled resources should be prevented. Because of this a mechanism to enforce a security policy exists in many OS. As a predominantly analysed subset of security policies used in operating systems AC policies exists.

At the functional side, a mechanism for enforcing AC policies should be able to enforce as many AC policies as possible. To achieve the desired effect, the implementation of such a mechanism has to follow the RM principles. With this the mechanism has to involved in every system operation that can be controlled by an AC policy and the mechanisms and it's components it depends on are protected against unauthorised manipulation. Also, they have to be sufficiently small and well-structured to be accessible to formal methods of correctness checking.

In the literature many proposals on how to build an architecture for enforcing AC policies can be found. Most of them have common elements, such as a set of PEPs and a PDP, but they do not define these components and their functionality in a common way. Also, many of them are not applicable for arbitrary AC policies and do not consider the specific issues – such as the environment – an implementation comes with.

To solve the issue of a missing, OS-specific RM architecture pattern, this thesis has defined the overall goal of defining such an architecture in a way that allows to enforce many kinds of AC policies and is clear enough to derive an implementation out of it. In order to do so, knowledge about how an RM can work must be obtained. Because of this, this thesis defines the first goal of developing the functional and non-functional requirements of an RM architecture. Also, the requirements of the policies to enforce must be known. Therefore, the second goal of this thesis was to analyse the requirements of state-of-the-art AC policies.

For achieving these goals, at first – in chapter 2 – knowledge from the literature is presented, that serves as basis of this thesis' results. Based on this, in chapter 3 the requirements of the interfaces of the proposed architectures are analysed by applying the methodology of analysing the contracts between the components of an architecture.

After that, in chapter 3 the requirements of DABAC polices are identified. This analysis was done by at first identifying the components required to enforce an DABAC policy. In the second step then the identified components are mapped to the components of an RM architecture. The third step investigates, how the components identified as necessary to enforce an DABAC policy influence the components of an RM architecture and the contracts between them.

Out of the requirements of the RM architectures and the requirements of DABAC policies, a new RM architecture was developed in chapter 4. This was done by

identifying the functionality an DABAC policy requires and what components identified in chapter 3 can provide this functionality. Based on this the interfaces of the components are defined in terms of their input and output as well as their pre- and postconditions.

When integrating components to enforce AC policies into an OS there are several things to consider. At fist, the properties of an RM have to hold for this subsystem. At second, two complementary and possibly conflicting non-functional requirements must be fulfilled. On one hand the implementation should be as performant as possible to not influence the overall system performance in a functional unacceptable way. On the other hand the implementation should be isolated in such a way that neither faults of the implementation can influence the operating system nor vice versa.

To evaluate if this is possible for the developed design and a state-of-the-art OS, the developed design should be implemented within a highly modular OS that is written in Rust and provides address space isolation. This was achieved as described in and chapter 5 by integrating the developed architecture into the Poplar OS kernel based on the requirements analysed in chapter 4. Thereby the integration was done in a way where Poplar is booting, and the provided applications can run even with an enforced AC policy. This integration is also done in a way that allows to enforce as many AC policies as possible.

It was also shown how the implementation fulfils the properties of an RM. Thereby it can be stated that the enforcement mechanism in involved in every system operation that can be controlled by an AC policy. However, the properties of being tamper-proofed and being small and well-structured to be accessible to formal methods of correctness checking are only partially fulfilled.

The property of being tamper-proofed is not completely fulfilled because of the type of isolation between the Poplar kernel and the RM. This is due to the complete integration into the kernel whereby isolation is only provided through the viability mechanisms of the Rust language and their enforcement at compile time.

The property of being small and well-structured to be accessible to formal methods of correctness checking is only partially fulfilled due to several dependencies. Thereby the first dependency to note is the Rust compiler, which cannot be trusted to work correct because of a missing formal verification With using Rust the second dependency is, that Poplar depends on some parts of Rusts standard library. The last dependency for analysing the RM architecture integrated in the Poplar kernel is, that the Poplar kernel depends on some parts of the Poplar standard library (which adds an indirect dependency to the Rust standard library).

Further Research. Depending on these achievements, there are several topics further research could focus on. One such topic could be how a better fulfilment of the RM properties within Poplar could be achieved. Based on this, the performance impact of such an integration should be evaluated. With the knowledge of the previous two topics it could also be explored how to integrate the proposed architecture in a more complex OS or how to integrate this architecture in software running a higher level – such as a middleware platform or an application.

APPENDIX A

Legends and Tables

This appendix chapter is to present legends and tables which are useful or necessary to understand the main part of this thesis but were not presented there. Therefore, section A.1 describes the meaning of the symbols used in diagrams and section A.2 present the full comparison of the AC enforcement mechanisms regarding their requirements.

A.1 Explanation of Diagrams

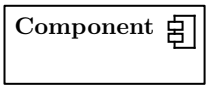
This section is to describe the symbols used in the diagrams and their semantics. Therefore, in section A.1.1 a legend for UML component diagrams and in section A.1.2 a legend for UML sequence diagrams is given.

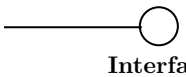
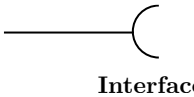
A.1.1 Explanation of UML Component Diagrams

According to [RQ12] a UML component diagram represents the elements of a system as *components*. It shows how these components are organized at runtime and the resulting dependencies.

Components are thereby a delimited system that can be accessed via clearly defined interfaces. The concrete realisation of a component can thereby be exchanged with another component without any changes on the overall system, if the components have the same interfaces.

The elements of a component diagram, which were used in section 3.2, are described below as in [RQ12].

Symbol	Name	Description
	Component	A component is a modular part of a system, which encapsulates the contents of the system and can be exchanged within its environment.
	Component with *Port*	A port is a classifier property that defines a unique point of interaction between the classifier and its environment or between the classifier's behaviour and its internal components.
	Interface	An interface is a type of classifier that declares a set of coherent public features and obligations. It specifies a contract that any instance of a classifier that implements it must fulfil.

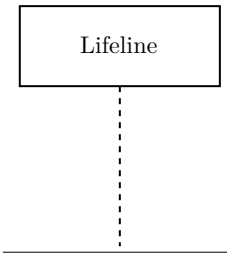
Symbol	Name	Description
	Provided Interface	This notation indicates that the classifier connected to this symbol provides this interface. An interface realisation is a specialised relationship between a classifier and an interface, signifying that the realising classifier conforms to the contract specified by the interface.
	Required Interface	This notation indicates that the classifier linked to this symbol relies on the specified interface to perform its function.

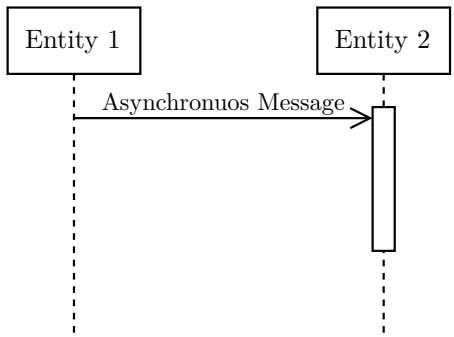
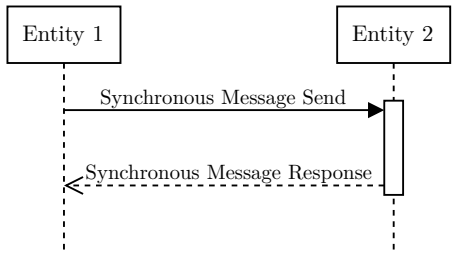
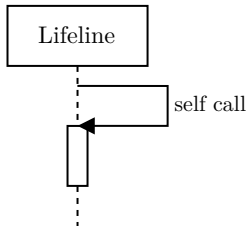
A.1.2 Explanation of UML Sequence Diagrams

According to [RQ12] sequence diagrams are showing the exchange of information between any communication partners within a system or between systems in general. They allow for the modelling of fixed sequences, temporal and logical sequence conditions, loops, and concurrency. Additionally, they provide insight into how communication occurs within the modelled system.

The primary element in a sequence diagram is the message. As per [RQ12], a message can represent the invocation of an operation, the response generated by the processing of an operation, a signal, a logical or analytical event, or the assignment of a value to a variable. According to [RQ12] a message "defines a particular communication between lifelines of an interaction. The message specifies not only the kind of communication e.g. raising a signal, invoking an operation, creating or destroying an instance, but also the sender and receiver." In chapter 4 messages are always used as either operation calls or responses. Messages were used to visualise synchronous and asynchronous communication as well as "self calls".

The elements of a sequence diagram, which were used in chapter 4, are described below as in [RQ12].

Symbol	Name	Description
	Lifeline	A lifeline represents an individual participant in the interaction. While parts and structural features may have multiplicity greater than one, lifelines represent only one interaction entity.

Symbol	Name	Description
	Asynchronous Communication	Asynchronous communication allows the sender to continue processing immediately after the send event without waiting for a response from the recipient. This means that the sender is not blocked after sending. Any response received at a later point in time is modelled as an asynchronous message.
	Synchronous Communication	Synchronous communication involves the sender waiting for the initiated behaviour to be completed at the receiver before receiving a response message and continuing processing. This approach not only implies that the synchronous message flow has ended, but also allows for the delivery of responses.
	Self Call	A self call is the act of an entity to send a message in the synchronous way to itself. This is used to model a local operation call.

A.2 Comparison of the Requirements of the AC Enforcing Mechanisms

This section presents the detailed comparison of the RM architectures from section 3.2. Thereby the term *PolIRM* (as a shorthand for *Policy Independent Reference Monitor*) is used for the architecture described in chapter 4.

All Architectures presented in section 3.2 share a Policy Enforcement Point. However, they differ in naming, communication with the other components as well as in pre- and postcondition and their additional functionality as follows.

Policy Enforcement Point (PEP)							
Architecture	Name	Caller	Callee	Input	Output	Preconditions	Postconditions
Flask	Object Manager (AC Decisions)	Subject	Object Manager (PEP)	Access Request	Indicator on allow and instance to use	—	Enforcing policy
	Object Manager (Attribution)	Subject	Object Manager (PEP)	Security Context	SID	Metadata changing allowed by policy	SID is assigned to object
	Object Manager (Cache)	Security Server (PAP)	Object Manager (PEP)	—	Indicator of finishing invalidation	AC policy changed	Cache invalidated
DFlask	PEP	Subject	PEP	Access Request	Indicator on allow	PDP reachable and secure communication possible	Enforcing policy

Architecture	Name	Caller	Callee	Input	Output	Preconditions	Postconditions
ExFlask	PEP	Subject	PEP	Access Request	Indicator on allow	PDP available	Enforcing policy
XACML	PEP	Subject	PEP	Access Request	Indicator on allow	—	Enforcing policy and execution of obligations on allow
PM	PEP	Subject	PEP	Access Request	Indicator on allow	—	Trigger to EPP
AppSPEAR	TOM	Subject	TOM	Access Request	Indicator on allow	—	Enforcing policy
PolIRM	PEP	Subject	PEP	Access Request	Indicator on allow	—	—

All Architectures presented in section 3.2 share a Policy Decision Point. However, they differ in naming, communication with the other components as well as in pre- and postcondition and their additional functionality as follows.

Policy Decision Point (PDP)							
Architecture	Name	Caller	Callee	Input	Output	Preconditions	Postconditions
Flask	Security Server (AC Decisions)	Object Manager (PEP)	Security Server (PDP)	Access Request	Indicator on allow and instance to use	—	—
	Security Server (Attribution)	Object Manager (PEP)	Security Server (PDP)	Security Context	SID	—	—
DFlask	PDP	PEP	PDP	Access Request	Indicator on allow	AC policy installed	—
ExFlask	PDP	PEP	PDP	Access Request	Indicator on allow	PIP reachable	—
XACML	PDP	Context Handler	PDP	Access Request	Indicator on allow	Access request in XACML format	—
PM	PDP (Normal Access Request)	PEP	PDP	Access Request	Indicator on allow	Operation is read or write	—

Architecture	Name	Caller	Callee	Input	Output	Preconditions	Postconditions
	PDP (Administrative Operation)	PEP	PDP	Access Request	Indicator on allow and success of execution	Operation is administrative operation	Operation is sent to PAP
	PDP (Policy Changing)	PAP	PDP	New policy	Indicator on success of change	Change allowed by policy	Changed policy stored
AppSPEAR	TPS	TOM	TPS	Access Request	Indicator on allow	—	Trigger to TEP on allow
PolIRM	PDP (Normal Access Request)	PEP	PDP	Access Request	Indicator on allow	—	Sending access request to EPP
	PDP (Normal Access Request)	PAP	PDP	Access Request	Indicator on allow	—	Sending access request to EPP
	PDP (Policy Changing)	PAP	PDP	New Policy	Indicator on success	Operation is allowed	New policy is stored

Policy Administration Point (PAP)							
Architecture	Name	Caller	Callee	Input	Output	Preconditions	Postconditions
Flask	Security Server (Policy Changing)	Subject	Security Server	Change Request	Change allowed by policy	New policy stored and inform object managers	
DFlask	—	—	—	—	—	—	—
ExFlask	PAP	Subject	PAP	Change Request	Indicator on success	Change allowed by policy	Changed policy stored in policy repository
XACML	PAP	Subject	PAP	Change Request	Indicator on success	Change allowed by policy	Changed policy stored in PRP
PM	PAP (Execution of administrative Operations)	PDP	PAP	Operation	Indicator on success	Operation is administrative	Changed state in EPP, PIP or PDP
	PAP (Request for Information)	PDP	PAP	Entity for which information are requested	Information from PIP	PDP request because of access request	—
AppSPEAR	—	—	—	—	—	—	—

Architecture	Name	Caller	Callee	Input	Output	Preconditions	Postconditions
PolIRM	PAP	Subject	PAP	Change Request	Indicator on allow and success	Change allowed by policy	New policy installed

Policy Information Point (PIP)							
Architecture	Name	Caller	Callee	Input	Output	Preconditions	Postconditions
Flask	—	—	—	—	—	—	—
DFlask	—	—	—	—	—	—	—
ExFlask	PIP	PDP	PIP	Entity for which information are requested	Information about this entity	Called from PDP	—
XACML	PIP	Context Handler	PIP	Entity for which information are requested	Attributes received from environment	Environment can process information request	—
PM	PIP (Information Lookup)	PAP	PIP	Entity for which information are requested	Information about this entity	—	—
	PIP (Information Changing)	PAP	PIP	Change Request	Indicator on success	—	Internal state of PIP altered
AppSPEAR	TEP (Information Lookup)	TPS	TEP	Entity for which information are requested	Information about this entity	—	—

Architecture	Name	Caller	Callee	Input	Output	Preconditions	Postconditions
PolIRM	PIP (Information Lookup)	PDP	PIP	Entity for which information are requested	—	—	
	PIP (Information Changing)	EPP	PIP	Changes to install	Indicator on success	—	Changes are installed
	PIP (Information Changing)	PAP	PIP	New information to install	Operation is allowed	New information are installed	

Event Processing Point (EPP)							
Architecture	Name	Caller	Callee	Input	Output	Preconditions	Postconditions
Flask	—	—	—	—	—	—	—
DFlask	—	—	—	—	—	—	—
ExFlask	—	—	—	—	—	—	—
XACML	—	—	—	—	—	—	—
PM	EPP (Event Processing)	PEP	EPP	Access Request	—	Access Request allowed	Changes in EPP or PIP
	EPP (State Modification)	PAP	EPP	Change Request	Indicator on success	Change allowed by policy	State of EPP changed
AppSPEAR	TEP (Event Processing)	TPS (PDP)	TEP (EPP)	Access Request	—	Access Request allowed	Changed policy state
PolIRM	EPP (Event Processing)	PDP	EPP	Access Request	Indicator on success	Access Request allowed	Changed information in PIP
	EPP (State Modification)	PAP	EPP	Change Request	Indicator on success	Change allowed by policy	State of EPP changed

Further Components							
Architecture	Name	Caller	Callee	Input	Output	Preconditions	Postconditions
Flask	—	—	—	—	—	—	—
DFlask	—	—	—	—	—	—	—
ExFlask	Policy Repository	PAP	Policy Repository	Change Request	Indicator on success	(PAP is caller)	Stored policies changed
XACML	PRP	PAP	PRP	Change Request	Indicator on success	PAP is caller	Stored policies changed
	Context Handler (AC Processing)	PEP	Context Handler	Access Request	Indicator on allow	Request in native format	Output in native format
	Context Hander (Information Request)	PDP	Context Handler	Entity for which information are requested	Attributes received from PIP	—	—
	RAP	PEP	RAP	Object, Operation and Parameters	Indicator on success	Access Request allowed	Operation performed

Architecture	Name	Caller	Callee	Input	Output	Preconditions	Postconditions
PM	RAP	PEP	RAP	Object, Operation and Parameters	Indicator on success	Operation is not administrative	Operation performed
AppSPEAR	—	—	—	—	—	—	—
PolIRM	—	—	—	—	—	—	—

Bibliography

[And72] James P. Anderson. Computer Security Technology Panning Study. Technical report, US Air Force Electronic Systems Div., Hanscom AFB, 1972.

[BBB+17] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System programming in rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 156–161, New York, NY, USA, 2017. Association for Computing Machinery.

[BLIZ20] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1–19. USENIX Association, nov 2020.

[BVZ17] Kevin Boos, Emilio Del Vecchio, and Lin Zhong. A characterization of state spill in modern operating systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 389–404, New York, NY, USA, 2017. Association for Computing Machinery.

[FAG11] David Ferraiolo, Vijayalakshmi Atluri, and Serban Gavrila. The policy machine: A novel architecture and framework for access control policy specification and enforcement. *Journal of Systems Architecture*, 57(4):412–424, 2011.

[FCHK16] David F. Ferraiolo, Ramaswamy Chandramouli, Vincent C. Hu, and D. Richard R. Kuhn. A Comparison of Attribute Based Access Control (ABAC) Standards for Data Service Applications. NIST Special Publication 800-178, 2016.

[FGHK05] David F. Ferraiolo, Serban Gavrila, Vincent Hu, and D. Richard Kuhn. Composing and Combining Policies under the Policy Machine. In *Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies*, SACMAT '05, page 11–20, New York, NY, USA, 2005. Association for Computing Machinery.

[FGJ14] David Ferraiolo, Serban Gavrila, and Wayne Jansen. On the unification of access control and data services. In *Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI 2014)*, pages 450–457, 2014.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, 1982.

[IBZ23] Ramla Ijaz, Kevin Boos, and Lin Zhong. Leveraging rust for lightweight os correctness. In *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification*, KISV '23, page 1–8, New York, NY, USA, 2023. Association for Computing Machinery.

[LCG+17] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 234–251, New York, NY, USA, 2017. Association for Computing Machinery.

[LSA+] Amit Levy, Leon Schuermann, Niklas Adolfsson, Pat Pannuto, et al. The tock book. https://book.tockos.org/. Last access on 27/02/2024 16:45.

[Mey92] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[MK14] Nicholas D. Matsakis and Felix S. Klock. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery.

[MM02] Richard Mitchell and Jim McKim. *Design by contract, by example*. Addison Wesley, Boston, MA, 1. printing edition, 2002.

[OAS17] eXtensible Access Control Markup Language (XACML) Version 3.0 Plus Errata 01: OASIS Standard incorporating Approved Errata, 2017.

[PBP13] Henrik Plate, Cataldo Basile, and Stefano Paraboschi. Policy-driven system management. In John R. Vacca, editor, *Computer and Information Security Handbook (Second Edition)*, pages 415–448. Morgan Kaufmann, Boston, second edition edition, 2013.

[PCO19] Andre Pinho, Luis Couto, and Jose Oliveira. Towards Rust for Critical Systems. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 19–24. IEEE, 2019.

[PYG00] Dimitrios Pendarakis, Dr. Raj Yavatkar, and Dr. Roch Guerin. A Framework for Policy-based Admission Control. RFC 2753, jan 2000.

[Red] Redox Developers. The redox operating system book. `https://doc.redox-os.org/book/`. Last access on 27/02/2024 16:00.

[RQ12] Chris Rupp and Stefan Queins. *UML 2 glasklar: Praxiswissen für die UML-Modellierung*. Hanser Verlag, München, 4th updated and expanded edition, 2012.

[SA23] Marius Schlegel and Peter Amthor. Putting the pieces together: Model-based engineering workflows for attribute-based access control policies. In Pierangela Samarati, Marten van Sinderen, Sabrina De Capitani di Vimercati, and Fons Wijnhoven, editors, *E-Business and Telecommunications*, pages 249–280, Cham, 2023. Springer Nature Switzerland.

[Sch21] Marius Schlegel. Trusted enforcement of application-specific security policies. In *Proceedings of the 18th International Conference on Security and Cryptography - SECRYPT*, pages 343–355. INSTICC, SciTePress, 2021.

[SSL+99] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The flask security architecture: System support for diverse security policies. In *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*, SSYM'99, page 11, USA, 1999. USENIX Association.

[Woo] Isaac Woods. The Poplar Book. `https://poplar.isaacwoods.dev/book/`. Last access on 27/02/2024 16:40.

Declaration of Originality

I hereby declare that this thesis is my own original work, which has been composed by me without using aids other than those specified. I have clearly referenced all sources, both published or unpublished, which have been directly or indirectly used in the work. This work has not been, previously or concurrently, submitted to any other examination authority.

Ilmenau, 17th May 2024

Eric Wintzler