Aalto University
School of Science

Master’s Programme in Computer, Communication and Information Sciences

Designing Kilpi, an Authorization Framework for Web Applications

Modeling and Implementing an Open Source TypeScript Framework

Jussi Nevavuori

Master’s Thesis
2025

Aalto University
School of Science

Author Jussi Nevavuori

Title Designing Kilpi, an Authorization Framework for Web Applications — Modeling and Implementing an Open Source TypeScript Framework

Degree programme Computer, Communication and Information Sciences

Major Software and Service Engineering — Service Design and Engineering

Supervisor and advisor Dr. Sanna Suoranta

Date 11 September 2025 **Number of pages** 77+7 **Language** English

Abstract

Designing and implementing an authorization system for a web application is a difficult task, which poses major operational and security risks, if constructed incorrectly. Due to the wide variety of existing authorization models and complexity of requirements for an authorization system, implementing such a system can be a challenging task. There is a lack of comprehensive open source solutions for JavaScript and TypeScript applications for authorization, with most solutions being enterprise-oriented languages, proprietary paid third-party services or platform-specific solutions.

This thesis introduces Kilpi, an open source TypeScript library for modeling and implementing authorization systems for web applications. This thesis discusses the goals, requirements and challenges of designing and implementing such a system, and evaluates the fully implemented and available Kilpi open source library against these requirements. The most important goals of Kilpi are developer friendliness and flexibility to suit most applications, use cases and authorization models. Additionally, Kilpi provides the flexible functional policy-based access control model based on defining policies as TypeScript functions.

This thesis evaluates Kilpi against guidelines from literature as well as tested in multiple production applications and finds it to fulfill its goals well as a flexible authorization solution. Future development includes primarily only superficial improvements on usability.

Keywords Authorization, Access Control, TypeScript, Open Source, Web applications, Kilpi

Aalto University
School of Science

Tekijä	Jussi Nevavuori
Työn nimi	Kilpi–autorisaatiokehyksen suunnittelu web-sovelluksille — Avoimen lähdekoodin TypeScript-kehyksen mallintaminen ja toteuttaminen
Koulutusohjelma	Computer, Communication and Information Sciences
Pääaine	Software and Service Engineering — Service Design and Engineering
Työn valvoja ja ohjaaja	Dr. Sanna Suoranta
Päivämäärä	11 September 2025
Sivumäärä	77+7
Kieli	englanti

Tiivistelmä
Web-sovelluksen autorisaatiojärjestelmän suunnittelu ja toteuttaminen on haastavaa, ja virheellinen toteutus voi aiheuttaa merkittäviä turvallisuus- ja operatiivisia riskejä. Autorisaatiomallien moninaisuuden ja autorisaatiojärjestelmän vaatimusten monimutkaisuuden vuoksi järjestelmän toteuttaminen on vaativa tehtävä. JavaScript- ja TypeScript-sovelluksille ei ole kattavia avoimen lähdekoodin autorisaatioratkaisuja, ja useimmat olemassa olevat vaihtoehdot ovat suurille yrityksille suunnattuja kieliä, maksullisia kolmannen osapuolen palveluita tai alustakohtaisia ratkaisuja.

Tämä työ esittelee Kilpi-nimisen avoimen lähdekoodin TypeScript-kirjaston, joka mahdollistaa autorisaatiojärjestelmien mallintamisen ja toteuttamisen web-sovelluksissa. Työssä käsitellään järjestelmän suunnittelun ja toteutuksen tavoitteita, vaatimuksia ja haasteita sekä arvioidaan valmista ja julkaistua Kilpi-kirjastoa näiden vaatimusten näkökulmasta. Kilpi-kirjaston tärkeimmät tavoitteet ovat kehittäjäystävällisyys ja joustavuus, jotta se soveltuu mahdollisimman useisiin sovelluksiin, käyttötapauksiin ja autorisaatiomalleihin. Lisäksi Kilpi esittelee joustavan funktionaalisen politiikkapohjaisen autorisaatiomallin, jossa käyttöoikeuksien linjaukset määritellään TypeScript-funktioina.

Tässä työssä arvioidaan Kilpi-kirjastoa kirjallisuudesta löytyvien arviointiperusteiden pohjalta sekä integraation pohjalta useisiin tuotantosovelluksiin. Näillä perusteilla Kilpi todetaan täyttävän hyvin joustavan autorisaatioratkaisun tavoitteet. Tuleva kehitystyö keskittyy pääasiassa pinnallisiin käytettävyyden parannuksiin.

Avainsanat Autorisaatio, Pääsynhallinta, TypeScript, avoin lähdekoodi, web-sovellukset, Kilpi

Preface

I want to thank Dr. Sanna Suoranta for her support, guidance and positive attitude. I also want to thank the open-source community and the BrisJS community in Brisbane, Australia for allowing me to present my work and for their feedback. I am grateful to all feedback, comments and interest shown towards my work by friends, colleagues and online communities.

Lastly, I want to thank my partner for supporting me through the process of writing this thesis. Thank you!

Otaniemi, 11 September 2025

Jussi Nevavuori

Contents

Abbreviations

ABAC	Attribute-Based Access Control
AC	Access Control
ACaaS	Access Control-as-a-Service
ACL	Access Control List
ACM	Access Control Matrix
ARBAC	Administrative Role-Based Access Control
B2B	Business-to-Business
B2C	Business-to-Consumer
B2E	Business-to-Employee
BLP	Bell–LaPadula Model
AuthN	Authentication
AuthZ	Authorization
CWE	Common Weakness Enumeration
DAC	Discretionary Access Control
DBMS	Database Management System
DoS	Denial of Service
DSL	Domain-Specific Language
DX	Developer Experience
EBAC	Entity-Based Access Control
FGAC	Fine-Grained Access Control
FGA	Fine-Grained Authorization
GEO-RBAC	Geographic Role-Based Access Control
HRBAC	Hierarchical Role-Based Access Control
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JS	JavaScript
LBAC	Lattice-Based Access Control
MAC	Mandatory Access Control
NIST	National Institute of Standards and Technology
NPM	Node Package Manager
OASIS	Organization for the Advancement of Structured Information Standards
OPA	Open Policy Agent
OrBAC	Organizational Role-Based Access Control
OWASP	Open Web Application Security Project
PAP	Policy Administration Point
PDP	Policy Decision Point
PerBAC[†]	Permission-Based Access Control
PEP	Policy Enforcement Point
PERM	Policy, Effect, Request, Matchers
PIP	Policy Information Point
PML	PERM Modeling Language
PolBAC[†]	Policy-Based Access Control

PurBAC[†]	Purpose-Based Usage Access Control
RB-RBAC	Rule-Based Role-Based Access Control
RBAC	Role-Based Access Control
ReBAC	Relationship-Based Access Control
RLS	Row-Level Security
ROBAC	Role and Organization Based Access Control
S-OrBAC	Service Organization Based Access Control
SaaS	Software-as-a-Service
SECaaS	Security-as-a-Service
SoD	Separation of Duties
SPA	Single Page Application
TBAC	Task-Based Access Control
TMAC	Team-Based Access Control
T-RBAC	Temporal Role-Based Access Control
TS	TypeScript
UI	User Interface
VBAC	View-Based Access Control
XACML	eXtensible Access Control Markup Language

[†] All conventionally referred to as PBAC. For clarity, and to avoid ambiguity, PerBAC, PurBAC, and PolBAC are used here instead.

1 Introduction

Authorization and access control are important components of all software systems. Their purpose is to secure applications by limiting the access of users to a software system in both what they are allowed to see and do [82]. This is especially true for web applications. The rise of cloud-native applications operating the sensitive data of multiple businesses and users on a shared multi-tenant platform [14] poses a real risk to both individuals and organizations in the case of security failures. The average cost of a data breach in 2021 was estimated to be 4.2 million US dollars [40].

The study of software authorization dates back to the 1970s with the first research efforts initiated by the U.S. Department of Defense [82, 47]. Since then, multiple different models and methods of authorization have been proposed over the years [82, 59]. Still, authorization remains a complex topic for modern web developers with broken access control having risen to be the number one issue in the Open Web Application Security Project (OWASP) 2021 top 10 list of web application security risks [69]. Similarly, the Common Weakness Enumeration (CWE) report lists four out of the top 25 most dangerous software weaknesses being related to authorization [43].

There is no standard way of implementing authorization in web applications. Alternatives range from custom-built authorization solutions and marginally used open-source libraries [65] to framework- and platform-dependent solutions such as Firebase Security Rules [32] or proprietary and paid third-party solutions such as Auth0 [11] or Permit.io [78]. The few standards include eXtensible Access Control Markup Language (XACML) and Open Policy Agent (OPA), however, their main adoption is in enterprise software [66, 17].

Designing a universal authorization library for Javascript (JS) and Typescript (TS) applications is a challenging task, given the increasing number of frameworks and web programming paradigms [92], the difficulty of implementing a good TypeScript developer experience, and the overall complexity of authorization in practice. However, when done correctly and as open source, a well-designed authorization library could positively impact a large number of web developers and applications. This thesis focuses on TypeScript, as any TypeScript library will work for JavaScript applications as well and TypeScript libraries can utilize TypeScript features to improve developer experience and safety when designed well.

1.1 Research Questions and Contributions

This thesis proposes Kilpi [61], an open-source authorization library intended for JS/TS applications. The aim of Kilpi is to provide a flexible and universal solution for authorization in web applications and other JS/TS applications, allowing developers to implement a proper authorization system into any application, independent of their framework or platform of choice and without enforcing a specific authorization model. Kilpi is published as an open-source library available on the NPM registry [62]. Additionally, this thesis presents the *Functional Policy-Based Access Control (FPBAC) model*, a generic approach as a superset of all other access control models, on top of which Kilpi is built.

Furthermore, this thesis will answer the following research questions:

1. What are the key challenges and requirements for implementing secure and flexible authorization in modern web applications?
2. How should a universal authorization library for JavaScript and TypeScript applications be designed to address these challenges and requirements?
3. How does Kilpi, the proposed solution address the requirements and challenges identified in the previous questions?

1.2 Methods

This thesis has utilized three primary methods to achieve its goals of understanding authorization and designing a solution for the identified challenges.

- **Literature review** of existing authorization models, methods and paradigms, to understand the concept, depth and breadth of authorization and access control, and the challenges and existing solutions and architectures common in authorization systems.
- **Community review** of existing open-source libraries, access control languages, third-party providers and other authorization solutions to understand the current state of authorization in JS/TS applications.
- **Iterative design and validation** of Kilpi. Kilpi has been designed to suit the needs of real and complex business requirements and has been validated in multiple production applications. The design and validation of a generic authorization solution have taken place over more than a dozen full-stack web applications, most of which power businesses in production. This has allowed me to design and validate Kilpi as a generic authorization framework.

1.3 Structure of Thesis

Chapter 2 provides a literature review and covers background information on authorization and access control, including prior research, key concepts, models, implementations, solutions, and challenges in the field. The chapter also defines the terminology used in this thesis. Chapter 3 motivates and introduces Kilpi, its architecture, design principles, and implementation details. Chapter 4 evaluates Kilpi as a solution in relation to existing literature, criteria and guidelines, but also by comparing it to existing solutions and evaluating it against its design goals and requirements. Finally, Chapter 5 will conclude the thesis, summarizing the findings and contributions of this work and discussing potential future work.

2 Authorization

In web applications, especially in increasingly common software-as-a-service (SaaS) applications, authorization is crucial for the functionality and security of the system. Authorization systems consist of different ways of making authorization decisions and policies determining which *subjects* are allowed to take which *actions* on which *objects* [59]. This includes both the systems used to define these policies and the mechanisms used to enforce them, as well as the behavior of the system in response to both granted and denied requests [82]. The different concepts and models of authorization systems are discussed in more detail in subsequent sections.

Authorization is a cross-cutting concern [49] as it is not limited to any single software module. Instead, it affects most parts of a web application. This poses a challenge especially for modern web services that are often distributed, heterogeneous, and highly dynamic [84]. Web applications range from highly interactive single-page and full-stack applications to more traditional server-side multi-page applications. This poses a challenge for any authorization system that attempts to work consistently across all different web programming paradigms, both on the client and on the server. An authorization system cannot be limited to only work on some parts of an application, due to its cross-cutting nature.

Authorization is by no means a trivial part of an application, rather, it can be one of the largest core concerns of the design and architecture of a software system. This is, for example, the case for the archive systems of large and renowned libraries [50]. Similarly, for cloud providers, designing a comprehensive authorization system is essential for cloud computing [6], due to the highly dynamic and distributed nature of the cloud environment.

2.1 Authorization or Access Control

The terminology behind *access control* and *authorization* is ambiguous, not standardized and varies between sources [59]. The terms "access control" and "authorization" are sometimes used interchangeably but are sometimes defined to mean different aspects of the same system of granting and restricting access [59]. Authorization may also refer to the result of an access control decision, i.e. the software may *grant an authorization*, making the caller *authorized* to perform the operation. This thesis, however, refers to these outputs explicitly as *decisions* or *authorization decisions* and *grants* or *denials*.

In this thesis, the terms "authorization" and "access control" are used interchangeably to refer to the mechanisms and models governing what actions users or systems are permitted to perform and what data they are allowed to access. No distinction is drawn between the systems and models used to define policies from those used to enforce them, unlike some sources [59].

The term "authorization" is preferred within the context of web applications in accordance with the modern web development community [63, 8, 33]. For this reason, it is also used as the primary term when referring to authorization and access control in this thesis. The term access control is used especially when referring to established terminology. Primarily, it is used in the context of established access control models such as Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC).

2.2 Authentication and Authorization

Authentication is the process of correctly and securely identifying a user, system or other entity accessing the application. It is separate from authorization, but also a prerequisite, since usually a system is unable to define what actions a user is allowed to perform without having first established their identity [82]. Any unreliability or unsecurity in the authentication system also directly compromises the authorization system that relies on the correctness of authentication [82].

Authorization systems may be able to utilize this dependence on authentication. Similarly to authorization, authentication is a cross-cutting concern [49] as it may affect all parts of a web application. An authorization system may treat authentication as a first-class citizen and introduce a level of coupling between its authorization and authentication systems, for example to allow unified mechanisms of authenticating and authorizing the user. It is, however, recommended to loosen the coupling between the two systems using an adapter pattern or similar approach allows the authorization system to not depend on a single authentication provider and may instead be used across a variety of different authentication solutions [5].

This thesis does not cover authentication in detail, but rather assumes authentication as a black box that can securely and correctly establish the identity of the subject.

2.3 Principles

Authorization systems often follow a set of principles that are guides to secure system design. Below are listed some of the most important principles common to most authorization systems.

2.3.1 Least Privilege

The principle of *least privilege* states that a subject should only be granted the minimum privileges necessary to perform its intended tasks [80]. When implemented correctly, a systemn can avoid both *underauthorization* and *overauthorization*. When a subject is underauthorized, they are unable to use the system to complete the required tasks due to lack of access. Conversely, overauthorization presents an operational or security risk, where subjects may be able to access resources or perform actions that they should not be allowed to.

2.3.2 Separation of Duties

Separation of duties (SoD) is a system design principle that aims to prevent fraud or error by ensuring that no single individual has control over all aspects of a critical process [59]. For example, a user should not be able to both initiate a payment and approve it. Partially, separation of duties is a problem of service and organization design, but it must also be accounted for in the design of an authorization system.

2.3.3 Open and Closed Policies

Authorization systems are commonly implemented using either *open* or *closed policies*. A system built on open policies grants access by default, unless explicitly prohibited by *negative rules*. Closed policies, on the other hand, default to denying access, unless explicitly granted by a *positive rule* [82].

Combining positive and negative rules is also possible, however it may introduce authorization conflicts and complexity, as a conflict resolution strategy must be defined and implemented [82]. A common resolution strategy for closed systems is to grant access if at the least one positive rule and no negative rules apply (and vice versa for open systems) [97]. However, if the conflict resolution strategy is not made explicit, conflicts may cause ambiguity, which may lead to vulnerabilities by mistake. Following the principle of least privilege [80], it is usually best to design a system based on closed policies by explicitly designing what users may access.

2.4 Concepts, Definitions and Terminology

There are several concepts that are shared by most authorization models. The three most important concepts in almost all authorization systems are the *subject*, the *object* and the *action* [59]. In addition to these concepts, many authorization models also include other concepts such as *attributes* or the *environment*. This section explores these concepts in more detail and establishes a common terminology used throughout this thesis. The terminology used to refer to the concepts below is more standard, however slight variations in terminology do exist across literature.

2.4.1 Subject

The *subject* is the active entity requesting access to a resource [59]. Typically in web applications, the subject is received from the authentication system and refers to the user of the application (due to this, the terms *user* and *subject* are sometimes used interchangeably). However, the subject is a more generic entity and may also represent another system, service, process, application or AI agent accessing the system. An unauthenticated user or entity is also a subject, as some applications may grant public unauthenticated access to some resources. For this reason, the more generic term *subject* is preferred.

Some sources also refer to the subject as an *actor* [45] or a *principal* [59], however other sources use the latter term to refer to any component that is part of the authorization process. To avoid confusion, the terms *actor* or *principal* are not used in this thesis.

2.4.2 Object

The object is the passive entity that the subject is trying to access or perform an action on [59]. In web applications, this is commonly a resource such as a database record, a file, a web page or an API endpoint. The object may also be another system or service that the subject is trying to access. The object may also be referred to as the *resource*.

2.4.3 Action

The action defines the activity the user is attempting to undertake on the object [59]. Commonly, actions are either `read`, `write` or `excecute` actions, with `write` often being sub-divided into `create`, `update` and `delete` operations. In web applications, especially for REST APIs, the current HTTP verb (e.g. `GET`, `POST`, `PUT`, `PATCH`, `DELETE`) may often be considered the action.

An action is, however, not restricted to these options and may be of any granularity from coarsely distinguishing between `read` and `write` operations to more fine-grained distinctions between operations such as `publish`, `restore from archive`, `like` or `follow`.

2.4.4 Decision

A decision is the result of the authorization process, either a *grant* or a *denial* of access to the requested resource or action [97]. The decision may be represented as a boolean value, or it may be an entity containing more information about the decision, such as the reason for the decision or the policy or rule that was used to make the decision.

2.4.5 Attribute

To allow for even more fine-grained access control, many authorization models also consider the attributes of different entities [59]. Most commonly authorization systems consider the attributes of the subject and the object. Systems that also consider attributes are generally more flexible and scalable [6]. Attributes are discussed in more detail in Section 2.5.6 on Attribute-Based Access Control.

For the subject, common attributes include the user's identifier, role, memberships in groups and organizations, assigned permissions and scopes, age, device information, email address, location, region or request metadata such as IP address or the user's web browser or operating system.

The attributes for objects vary even more based on the type of object. Files may contain attributes such as the file type, size, owner, creation date, modification date and access permissions. Database records may have attributes such as the record type, status, owner or creation date.

The types of attributes considered in a system are primarily dictated by the requirements of the system and the authorization model used. Authorization models may in practice include any required data as attributes, such as even the user's current emotion to prevent angry users from performing critical actions [59].

2.4.6 Environment and Context

On top of considering the subject, object and action, some authorization models provide even more fine-grained control by considering the *environment* in which the request is made. The environment is usually characterized by one or more relevant attributes, such as the current time, region and local legislation, or application state. This, for example, allows for stricter access control in countries with stricter data protection laws, or restricting access based on the age of an object or the current time of day.

The environment is one part of the current *context* in which the request is made. Depending on application requirements, the context may also include additional information such as historical data, risk or urgency [59, 39].

The environment and context cover all relevant information considered in authorization decisions that is not either part of the subject, the object or the action or their attributes.

2.4.7 Policy and Policy-Based Access Control (PolBAC)

An authorization policy is an artifact specific to the system that defines the rules and requirements on access control decisions on any level of granularity and in accordance to the chosen authorization model [59, 93]. In centrally managed authorization systems, policies could be defined as textual documents acting as guides for system administrators. However, this thesis will primarily focus on policies defined using code or a domain-specific language (DSL), such that the artifact itself is either an executable part of the program or may be executed as input to an authorization engine. Examples of simple policies are listed below.

– A user may delete a file if they are authenticated as the owner of the file.

– A post may be viewed by any subject if marked as public, otherwise only by the author and their friends.

– A user may update a record if they are its owner or have been granted "editor" level permission on the record.

In practice, policies can often be much more complex and involve multiple conditions, consider different types of subjects and include domain-specific constraints and business logic. This thesis simply refers to these artifacts as policies, however multiple terms such as *authorization policy*, *access policy*, *access rule*, *policy rule* and *access control policy* are also commonly used synonymously [59].

Policy-Based Access Control (PolBAC) refers to an access control model based on a *policy language* used to define policies as conditions of the subject, action, object and their attributes to output an authorization decision [97]. This thesis uses the term Policy-Based Access Control or PolBAC to refer to any access control model that makes decisions based on an evaluation of a policy or policies defined as functions of the form $f : X \rightarrow Decision$, where X is the set of all inputs given to the system (e.g.

subject, object, action, attributes, environment, etc.). This is in contrast to *permissions* discussed next in Chapter 2.4.8.

2.4.8 Permissions and Permission-Based Access Control (PerBAC)

A permission is an object or an entity stored in the system (e.g. in the file system, database or in-memory), whose existence grants access to the subject, object, action or other parameters defined in the permission [38]. Denied access is represented by non-existence of a permission. This covers multiple authorization model discussed in Section 2.5 [59, 6, 39].

Some systems also implement *prohibitions*, the inverse of a permission, whose existence similarly denies access to the attributed resource. As discussed in Section 2.3.3 on open and closed policies, the existence of both permissions and prohibitions (negative and positive rules) requires a resolution strategy for conflicting rules.

Authorization models can be classified as *Permission-Based Access Control (PerBAC)* if they are based on managing and validating permission (and prohibition) objects in the system. Compared to policy-based access control, permission-based systems may be more rigid as permissions are static objects that must be managed manually or via automations, compared to dynamically evaluated policies. Permission-based systems do, however, offer benefits not present in policy-based systems, such as the ability to query the system for all permissions a subject has, or all subjects that have access to a specific object.

All models can be roughly divided into policy-based models and permission-pased models, based on whether access is granted by the existence of related permission and prohibition objects or by evaluating a policy function. The two approaches are not mutually exclusive and can be used together. A system may, for example, implement permissions as well as policies which in addition to other factors, consider the existence of these permissions.

2.5 Authorization Models

Over the years, multiple different authorization models have been developed to address the growing complexity and requirements of authorization in software systems. The following sections explore the most common and relevant authorization models for web applications. There does not exist a universal model of authorization to fit the requirements of all applications, rather the needs of different systems are satisfied by applying the correct authorization models [35].

Some of the models have been designed to overcome the limitations of earlier models and to extend their capabilities, whereas other models stem from a fundamental reinterpretation of authorization [80]. Most models, however, originate from new requirements and opportunities, as organizations, technologies and practical needs evolve [80].

Only models that are relevant to web applications are discussed here. Access control models designed specifically for other environments, such as *View-Based Access Control (VBAC)* for relational databases [48], are omitted.

2.5.1 Discretionary Access Control (DAC)

The *Discretionary Access Control (DAC)* model is one of the simplest and oldest authorization models [59]. It relies on the identity of the subject and ownership of objects, where only the owner and those explicitly granted access by the owner can access an object [6]. By default, the creator of an object is granted ownership and thereby full and exclusive access to it. It is — as the name suggests — at the owner's discretion to grant access to other subjects. Thus, in DAC, all access control is fully controlled by individual users. While DAC is suited for distributed systems without centralized control over authorization, such as a file system, it is also very coarse and not able to enforce fine-grained policies. DAC can also be implemented for groups of subjects instead of individual subjects directly [81].

DAC is commonly implemented using an *Access Control Matrix (ACM)*. Proposed in 1971 [59], ACM is one of the earliest and simplest authorization implementations [82, 48]. It is a matrix ACM where each column represents an object $o_i \in O$ in the system and each row a subject $s_i \in S$ (with S and O respectively representing the set of all subjects and objects in the system). The intersecting entry contains the set of allowed actions $ACM[s, o] \subseteq A$ for the subject on the object, out of all allowed actions A, e.g. *read* or *write* ($A = \{r, w\}$). This is illustrated in Figure 1.

	o_1	o_2	o_3
s_1	{r}	{r, w}	$\varnothing$
s_2	$\varnothing$	{r}	{w}

Figure 1: Example of an Access Control Matrix ACM.

In practice the matrix is often very sparse and therefore not stored as a matrix directly [59]. More commonly, depending on the context and requirements of the application, it is implemented using one of the following approaches:

- An *Authorization Table* stores all non-empty entries of the matrix as a set of tuples containing the subject, object and action [59].
- An *Access Control List (ACL)* stores the list of subjects and their allowed actions for each object separately [82].
- A *Capability List* is the inverse of an Access Control List, where instead of storing the non-empty cells of the Access Control Matrix per object, they are stored per subject [82].

2.5.2 Mandatory Access Control (MAC)

Mandatory Access Control (MAC) is a simpler, stricter and more secure model compared to the Discretionary Access Control model [37]. The system consists of subjects and objects, with labels representing e.g. security clearances or integrity attached to them. The labels are only controllable by a centralized system management authority. Users have no control over the system. The labels are used to decide access based on model-specific rules [59]. MAC, controlled by a centralized authority, can be thought of as the opposite of DAC, in which access control is fully distributed to the subjects.

Mandatory Access Control models were originally common in government and military organizations [80], as they are well suited for more rigid and centralized organizations with high security requirements. Mandatory Access Control has been standardized in the *Orange Book of the U.S. Department of Defense*, however in practice even the military often finds MAC too restricting [82]. Although simpler and often more secure, MAC is not flexible and does not provide fine-grained control over access. A centralized administration can also not become an organizational burden and a single point of failure [46].

MAC is not a model itself, but refers to a group of different formalized models for different purposes. The list below contains the most common MAC models.

– **Bell–LaPadula Model (BLP)**

  A model designed to ensure confidentiality and secrecy [15, 80], especially in government and military applications. It labels all objects and subjects with one of the security labels *Top Secret*, *Secret*, *Confidential* and *Unclassified* [59]. BLP then follows the *No-read-up* and *No-write-down* rules, such that subjects can only read objects at or below their own security label, and write to objects at or above it. This ensures that no confidential information can be leaked to lower security levels [37].

– **Biba Model**

  The Biba model is similar to BLP, but with inverse rules designed for ensuring integrity [59]. Unlike confidentiality, which is concerned with who information can be disclosed to, integrity is concerned with the correctness and trustworthiness of information [37]. The model uses multiple labels from high to low integrity and uses the *Read-up* and *Write-down* rules to ensure that high integrity subjects do not read low integrity objects, and that low integrity subjects do not modify high integrity objects [59].

– **Lipner Model**

  The Lipner model is a model, which combines both BLP and Biba for commercial purposes, in order to preserve both integrity and confidentiality [59]. In commercial applications, both aspects are equally important in many cases, e.g. in payroll systems.

– **Clark-Wilson Model**

  The Clark-Wilson model is designed to prevent fraud in commercial applications by enforcing all access to objects to be mediated by *transformation procedures* and a set of four *enforcement rules* and five *certification rules* [59].

- **Chinese Wall Model** (or the *Brewer-Nash Model*)

  The Chinese Wall is model designed to prevent conflicts of interest [59], such as in consultancies working with banks and insurance companies, where employees have access to confidential information from competing clients. The model groups companies into conflict-of-interest classes and allows any subject to only access the data of a single company within each conflict-of-interest class.

2.5.3 Role-Based Access Control (RBAC)

For commercial and industrial organizations, neither DAC or MAC are optimal models. DAC assumes that users own the data, whereas in most organizations, it is the organization that owns the data. Moreover, DAC is often unsafe for these organizations [28]. Similarly, the rigid security classifications of data and subjects of MAC do not translate into the real needs of commercial and industrial organizations.

The *Role-Based Access Control (RBAC)* model was developed for this purpose, to better suit the needs of industrial and commercial organizations and civil governments through the use of centralized access control management [28]. In RBAC, organizations consist of different roles, such as *manager*, *employee*, *HR* or *accountant*. The access to organization resources is based on these roles [80]. Subjects are then assigned one or more roles, through which they access the system.

This more closely reflects organizations, where access is granted by role, e.g. a manager can access all employee records, while an employee can only access their own record, not by individual employees. In addition, changes common in organizations, such as promotions and hires are easier to perform, through simple reassignment of the user's roles [82]. Additionally, RBAC is often a simpler form of security for organizations [82].

The simplest implementation of RBAC is a variation of the Access Control Matrix discussed above, with the subjects replaced by roles [80]. In addition, there must also exists a second relation mapping users to their assigned roles. This structure is depicted in Figure 2.

RBAC is also well-suited for following the principle of least privilege. While not strictly enforcing it, RBAC makes it easier to implement by designing the roles carefully and assigning only minimal required roles to subjects [28, 82, 80]. Similarly, separation of duties can be implemented using RBAC [59]. This can be achieved statically by designing mutually exclusive roles and enforcing that any subject may not have two or more mutually exclusive roles assigned to them at once [82]. Alternatively, separation of duties can also be implemented more dynamically, depending on the implementation [59].

2.5.4 Extensions of RBAC

RBAC is simple and powerful, but for many real-world use cases it often needs extensions to account for different organizational needs. This section covers simpler extensions on top of the base RBAC model, such as hierarchies and administrative RBAC.

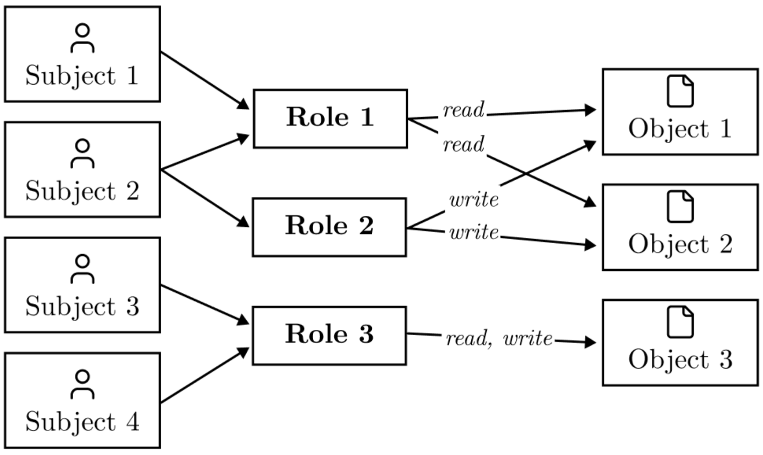

Figure 2: Simple RBAC model.

A common variation of RBAC is *Hierarchical Role-Based Access Control (HRBAC)*, where roles exist in a hierarchy and inherit permissions from their parent roles [46, 82, 42]. For example, the role of *supervising engineer* inherits all permissions from its parent role of *engineer* [82]. Role hierarchy is, however, not dictated by organizational hierarchy, as a hospital director should not inherit the permissions of a physician despite being above all physicians in the hierarchy of the hospital [39].

Other variants extend the concept of roles to also apply to objects via *classes* [82] or *views* [39] by applying one or more classes to all objects. A class or a view is an abstraction that defines the type of object, e.g. a file, a patient record or an invoice. This can be further used to simplify access control management by only defining the interaction between roles and classes instead of roles and each individual object. Some models also permit a hybrid approach, where access to objects can be determined both on the class and instance levels.

Some implementations of RBAC consider *sessions* by allowing subjects to only have a subset of their roles selected as *active roles* within a single session [28, 39]. This allows the system to more granularly adhere to the principle of least privilege, by limiting the roles to only those required for the current task. Other implementations apply *constraints* to RBAC, such as mutually exclusive roles, in order to enforce separation of duties.

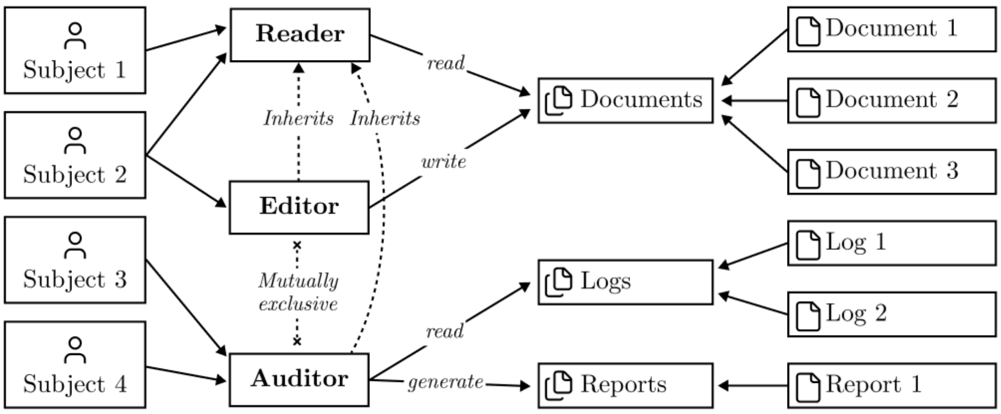

Figure 3: Example of RBAC with role hierarchies, constraints and object classes.

A common variation of RBAC is to also introduce a symmetrical but separate Administrative RBAC (ARBAC) system alongside the regular RBAC system. The ARBAC system works on the same principles as RBAC, but instead of managing user access, it manages administrative access to users, roles and permissions. A full RBAC model with sessions, role hierarchies and constraints alongside a parallel ARBAC system is illustrated in Figure 4.

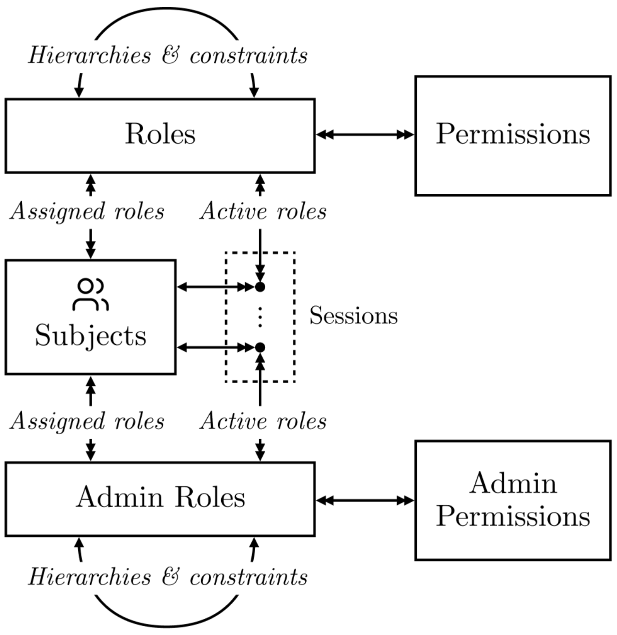

Figure 4: Parallel RBAC and ARBAC models with sessions, hierarchies and constraints.

Task-Based Access Control (TBAC) is another variation that introduces the concept of a *task* [48]. A task can be thought of as a "sub-role", and defines which subset of permissions the user has active based on the current objective. The subject's current role and task together define their access within the system.

There are also extensions that limit access to roles based on environmental factors. One of these is *Temporal Role-Based Access Control (T-RBAC)* that applies the concept of *time* by limiting the periods during which subjects have access to specific roles [59]. Similarly, *Geographic Role-Based Access Control (GEO-RBAC)* introduces the concept of *location* by restricting roles based on the subject's location [59]. These variations are not mutually exclusive and depending on organizational needs, multiple extensions of RBAC can be used simultaneously.

The first model that introduces collaboration to RBAC is *Team-Based Access Control (TMAC)* [89], which introduces a *team* as a layer of abstraction. TMAC was initially designed for a clinical context, in which the individuals and teams responsible for managing patient care are dynamic and change often, and whose requirements can not be satisfied by RBAC as it lacks dynamic roles and fine-grained permissions [89]. On top of regular RBAC, users can belong to teams with specific *team roles*, which can be dynamically permitted to access the required object classes and instances. Teams and their members can be dynamically re-assigned based on the context. It is a form of "just-in-time" permissions, for dynamic collaboration with minimal administrative overhead [89].

2.5.5 RBAC with Organizations

The previously discussed RBAC models do not consider the concept of *organizations*, although roles themselves are already an organizational concept [89]. The models discussed in the previous subsection, however, imply that they exist within the bounds of a single organization and do not account for multi-tenancy. The previous models are thus more suitable for *B2E (Business to Employee)* applications, whereas the following models are better suited for *B2B (Business to Business)* and *B2C (Business to Consumer)* applications [96].

The use of multi-tenant applications simultaneously used not only by multiple users but by multiple organizations as well is increasingly common, especially in cloud computing and modern SaaS applications. This subsection covers extensions of RBAC that apply the concept of organizations and multi-tenancy.

A simple and common solution for B2C and B2B applications is *Role and Organization-Based Access Control (ROBAC)* [96]. ROBAC extends RBAC by introducing organizations and organization roles. Instead of assigning roles to subjects directly, they are assigned organization-role pairs, which grant access to a specific organization under that role. A subject may even be simultaneously authorized to multiple organizations under different roles. This structure is shown in Figure 5. Access is granted by organization-role pairs, similarly to regular RBAC, but by requiring a role to exist from an organization-role pair that corresponds to the organization of the object. Each organization may manage which roles have which permissions individually. Due to slight variations in role names and permissions from organization to organization, the number of roles required to facilitate these changes with regular RBAC would grow too large, whereas ROBAC allows for this scalability.

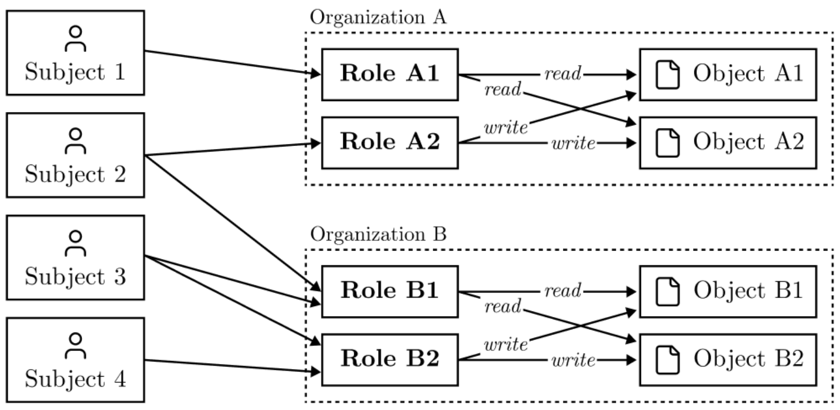

Figure 5: Role and Organization-Based Access Control (ROBAC) model.

However, ROBAC is limited in cross-organization access control and does not facilitate the sharing of resources across multiple organization. The *Multi-Tenant Role-Based Access Control (MT-RBAC)* model has been developed for cloud-based multi-tenant environments, where access may be shared to other organizations [88]. It recognizes different organizations as *issuers* consisting of one or more *tenants*. For example, an enterprise company can be considered an organization or an issuer, whereas its different departments, e.g. accounting, development or HR, may be recognized as its tenants. Each subject is assigned to a single tenant and has roles that grant access to permissions, both of which also belong to that tenant. The core idea of MT-RBAC is that tenants can *trust* other tenants, thereby allowing the tenants of the trustee to be assigned to the roles of the truster and access the resources of the trusting tenant [88]. This allows for example outsourcing development by establishing trust from the development tenant to a tenant of an outsourcing company. The model and tenant-trust based role assignation can be further modeled in more detail and in varying levels of granularity [88]. The core structure of MT-RBAC is shown in Figure 6.

Another multi-organization model is *Organization-Based Role-Based Access Control (OrBAC)* [39]. Unlike the much simpler ROBAC model, OrBAC introduces a more complex and formal structure. Similarly to ROBAC, organizations in OrBAC also *employ* subjects as members of an organization under a certain role. This abstracts the more concrete subjects to more abstract roles. Similarly, OrBAC *considers* different actions as *activities*, and *uses* different *objects* as *views*. All *employments*, *considerations* and *uses* are mappings of the concrete subject-action-object layer to the abstract role-activity-view layer [48].

Additionally, a *context* can be *defined* for a subject-action-object-organization tuple to contain additional information on the current task, such as urgency [39].

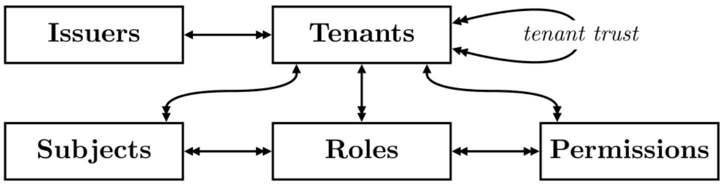

Figure 6: Multi-Tenant Role-Based Access Control (MT-RBAC) model (omitting details such as role hierarchies, constraints and sessions.

Permissions are then granted on a role-activity-view-context basis. The structure of OrBAC is shown in Figure 7.

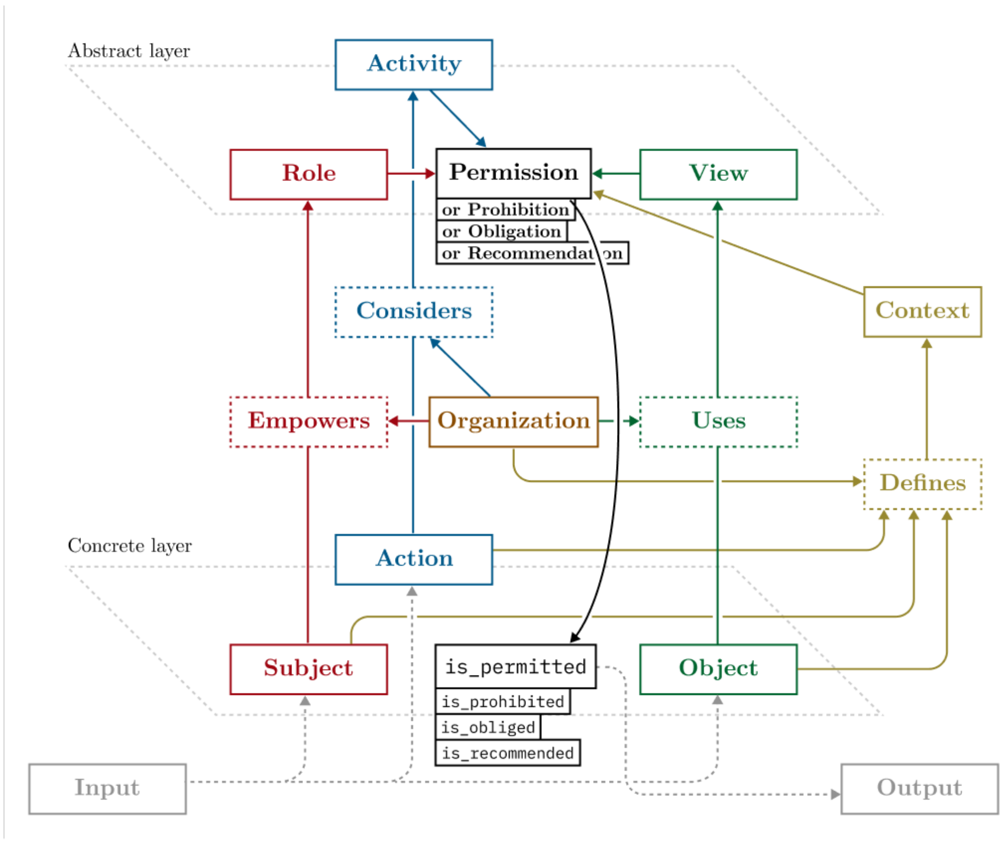

Figure 7: Organization-Based Role-Based Access Control (OrBAC) model.

OrBAC also supports an organizational hierarchy, as organizations themselves can act as subjects and can therefore be *employed* to another organization with a specific role [39]. This allows for modeling complex and hierarchical organizations. On top of granting permissions ("positive rules"), OrBAC also supports the same mechanism for *prohibitions* ("negative rules") as well as *recommendations* and *obligations*, which

— although not strictly part of access control — define other actions that the subject is respectively either recommended or required to perform [48]. The combination of positive and negative rules, however, creates the risk of conflicting rules that may introduce ambiguity without a clear resolution strategy. A common strategy is to grant access if and only if the policy yields one or more permissions and exactly zero prohibitions.

OrBAC can still be extended, such as by *Service Organization-Based Access Control (S-OrBAC)* [47] that introduces the concept of *services* to OrBAC. Permissions and prohibitions are also then related to a service. S-OrBAC allows for even finer granularity and more flexibility and critically allows organizations to expose their free or paid services to subjects external to the organization, with proper access control [47].

2.5.6 Attribute-Based Access Control (ABAC)

A fundamentally different take on authorization from RBAC is *Attribute-Based Access Control (ABAC)* that is based on *policies* or *rules* which consider the different *attributes* of the subject, object and the *environment* [95]. Traditional models, such as DAC, MAC and RBAC, which rely only on the identity of the object, are often not adequate for securing dynamic and distributed web applications, in which all parameters must be authorized with fine-grained access control [84]. ABAC is a safer and more flexible and fine-grained model than any model discussed thus far [34, 6]. For these reasons, ABAC has been the fastest growing access control model in recent years [34].

ABAC is a form of Policy-Based Access Control (PolBAC), as discussed in Section 2.4.7 and it is implemented with dynamic policies. The policies receive the subject, object and action and environment and have access to their attributes via *attribute authorities* or *attribute repositories*. The flexibility of ABAC is thereby only limited by the available attributes and computational resources [34].

A policy can be defined as a function of the subject s, action a, object o, environment e and their respective attributes A_s, A_o and A_e, which outputs a *decision*.

$$f : \{s, a, o, e, A_s, A_o, A_e\} \mapsto \text{Decision}$$

Let's consider the following example. A movie streaming service categorizes subjects by subscription status as either free or premium, and by age as 21 or older, 13 or older, or under 13. Movies are also categorized by their rating as either G (everyone), PG-13 (13 and older) or R (21 and older) and by release status as either new or old. To watch movies, the user must have an appropriate age for the the rating of a movie and new releases are only available to premium users. This simple system already introduced $3 * 2$ roles ($<$ 13 free user, $>=$ 21 premium user, ...) and $3 * 2$ permissions (Old G-rated movie, New R-rated movie, ...), totalling $6 * 6 = 36$ possible relations between roles and permissions to manage, even after limiting the range of the age attribute to only three distinct categories. Instead, ABAC can express this with a single policy, illustrated below as pseudocode:

```
can_watch_movie := function (subject, object) {
  (object.release == NEW AND subject.tier == PREMIUM) AND (
    (object.rating == R    AND subject.age >= 21) OR
    (object.rating == PG13 AND subject.age >= 13)
  )
}
```

In general, as the number of attributes and the range of their possible values increase, the number of roles, permissions and their relationships in RBAC grows exponentially. This makes RBAC impractical for such fine-grained access control. Additionally, if any attribute's range of values is infinite, the number of roles or permissions also becomes infinite, making RBAC theoretically impossible to implement.

To overcome these limitations, RBAC systems may implement *dynamic rule based role assignment*, allowing for more flexible roles [95]. These are known as *Rule-Based Role-Based Access Control (RB-RBAC)* systems, however they increase the complexity of RBAC without being able to provide the full flexibility of an attribute-based system. Furthermore, this does not even account for the fact that most RBAC systems are unable to consider environmental attributes. Many types of rules, such as allowing a physician to update the records of a patient only if they are the attending physician of the patient, are difficult to express in RBAC systems, requiring a unique role for each patient [95].

RBAC, all of its extensions, and even DAC and MAC can be seen as a subset of ABAC [59]. The subject's role (and even the organization or other attributes considered in the extensions) can be thought of as attributes, with the formal rules of RBAC being implemented using policies. Therefore, an ABAC system can be designed to also implement RBAC or any of its extensions, while still providing far more flexibility.

In practice, web applications implement a *Policy Enforcement Point (PEP)*. Usually, this is where the execution of the program stops to check the subject's access and either continues execution if the subject is authorized, or halts by returning an error for unauthorized access. The PEP then queries the *Policy Decision Point (PDP)* that evaluates whether the user is authorized or not. The PEP has access to the policies of the system, as well as all subject, object and environment attributes from attribute authorities [95]. This model is shown in Figure 8.

The attribute repositories are an abstraction over any method of providing data to the ABAC system. Commonly, attributes may be accessed from a dedicated attribute authority service, which manages all required attributes for the system. The policy decision point can then *pull* data from these services as required. However, the policy enforcement point may also *push* data by providing it directly to the policy decision point, eliminating the latencies of fetching attributes and complexity of maintaining separate attribute services. The push and pull mechanisms can also be combined, for example by pushing data that is always required for the policy, and pulling data that is only conditionally needed. The evaluation and computation of the policy including all data fetching is one of the performance bottlenecks in ABAC systems [86].

An important aspect of ABAC is the accuracy, integrity and availability of the

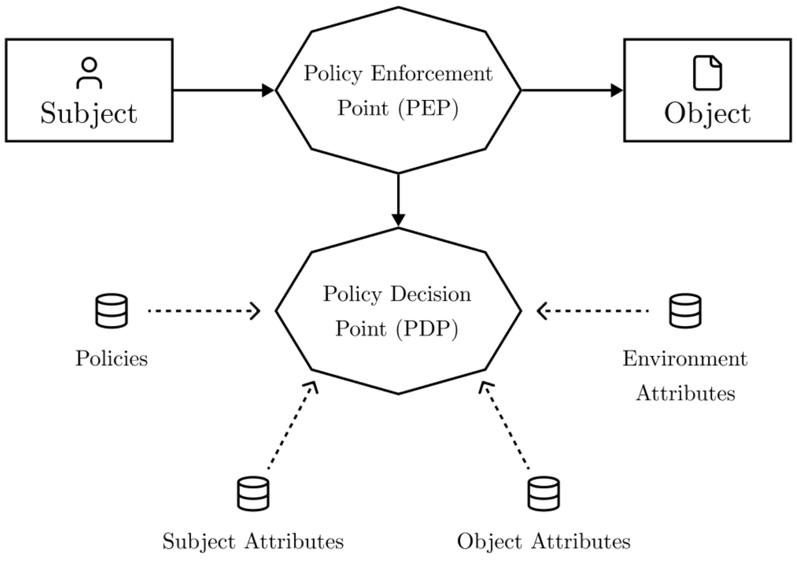

Figure 8: Attribute-Based Access Control (ABAC) model

attributes [34]. They need to reflect updates to the attributes quickly, and the attributes must come from a reliable source using secure communication, to ensure the integrity of the attributes and the access control system as a whole. The attributes themselves can be stored or computed based on stored values, or they may be human-generated or automatically assigned.

Although managing ABAC rules and attributes may create administrative overhead, especially in enterprise settings [34], ABAC can also simplify managing access. ABAC systems only rely on the attributes, therefore upon a creation of a subject or an object, only the required attributes need to be assigned [34]. There is no need to manage the relations between all identities of objects and subjects or roles. This is especially beneficial in dynamic environments, such as web applications, where the subjects and objects are not known in advance. The system is also dynamic, as changing any subject or object attributes automatically updates the subject's access.

2.5.7 Relationship-Based Access Control (ReBAC)

Relationship-Based Access Control (ReBAC) were created to overcome the limitations of RBAC and to allow basing authorization on the relationship between entities [59], which is not possible in RBAC. ReBAC systems are very common especially in increasingly popular social network applications but also in other types of applications [30]. Examples of common types of policies that rely on the concept of relationships include

– *Social relationships* such as friendships or family (e.g. allow friends of subject to send messages).

- *Professional relationships*, such as physician-patient, lawyer-client or teacher-student (e.g. allow only lawyer to see client's files).
- *Organizational relationships*, such as manager-employee, contractor-company or department-company (e.g. allow manager to view employee's performance reviews).

The relationships can be any graph structure, such as undirected graph of friend-to-friend relationships (such as in Facebook), a directed graph such as follower-to-followee (such as in Twitter) or a hierarchical graph such as organizational structures. The relationships can be manually defined, such as the "joint consent" friendship relations defined on Facebook by individual users, or automatically extracted from data, such as in Google Buzz [29].

Relationships are typically graphs $\mathcal{G}(V)$ between vertices $v \in V, V = \{S, O\}$ where S is the set of all subjects and O the set of all objects, which define directed relationships as pairs $(v_a, v_b) \in \mathcal{G}(V),\ v_a, v_b \in V$ [29]. These relationships define *assertions*, such as `manager(Alice, Bob)` asserting that Bob is the manager of Alice [29]. Policies are defined based on these assertions.

Many systems however define multiple types of relationships $\mathcal{I}$, such as friendships, colleagueships and memberships simultaneously [29]. This extends the previous graph to $\mathcal{G}(V, \mathcal{I})$ and extends the relationships tuples to include the type of relationship $i \in \mathcal{I}$ as $(v_a, v_b, i) \in \mathcal{G}(V, \mathcal{I}),\ v_a, v_b in V, i \in \mathcal{I}$. This allows for a wide variety of policies, with the type of relationship even being able to include more contextual information such as distinguishing the relationships of *consulting physician of patient* and *treating physician of patient* instead of simply *physician of patient* [29]. Relationships are also composable, such as defining the relationship *friends of friends* from the existing friends relationship, common in social applications [29].

A common implementation of ReBAC is the Google-proposed Zanzibar system, which provides a unified model and policy language for globally distributed and highly performant Relationship-Based Access Control, which powers systems such as Google Drive, Google Cloud and YouTube [77].

2.5.8 Entity-Based Access Control (EBAC)

ReBAC does not support fine-grained access control at the attribute level [59]. This is where *Entity-Based Access Control (EBAC)* comes into play. It addressess the limitation of both ReBAC, which cannot account for attributes, and ABAC, which cannot account for relationships. This is achieved by combining ReBAC and ABAC into a single EBAC system that can consider both relationships and attributes simultaneously for highly fine-grained access control [59].

Entity-Based Access Control considers all subjects and objects as *entities* with attributes similarly to ABAC, and complex relationships similarly to ReBAC [16]. Policies are then based on both the attributes as well as the relationships.

An implementation of EBAC is the Auctoritas system and policy language, which was initially designed as an alternative to the ABAC-based XACML [16, 50].

Auctoritas was found to perform better than XACML, on top of being able to express composed relationships (friends of friends) and other features, not representable in XACML. The Auctoritas language however suffers from a lack of popularity and adoption in practice [59].

2.5.9 Hybrid Models and Extensions

The authorization models mentioned above do not form an exhaustive list but rather attempt to cover the most common models and their variations. Many more models exist, such as Emotion-Based Access Control [59], which considers the user's current emotions, Purpose-Based Usage Access Control (PurBAC), which considers *why* an object is being accessed [46] as well as other models that account for risk, completed trainings, time or historical data [46].

In practice, all applications do not strictly need to follow a single authorization model. Applications may consist of different subsystems, each of which must use a different authorization model. For example a social network such as Facebook might apply DAC and ReBAC for the sharing of user-generated content, but ROBAC for managing groups, and RBAC with ARBAC for system administration.

Furthermore, an application does not even have to strictly stick to a single formally defined authorization model. Take for example the complex OrBAC model. If the requirements of the application state so, implementing parts of the OrBAC model such as context or allowing organizations to act as subjects is not required. Omitting or modifying parts of formalizations, or adding any extensions does not invalidate an authorization model, but rather alters it to suit application requirements. As an example inspired by T-RBAC and GEO-RBAC [59], the concept of time and location may be added to any model if the authorization system permits accessing these attributes.

All hybrid and custom authorization models are valid, if they secure the system and fulfill the authorization requirements of the system. On top of validating an authorization model to fulfill the system requirements, models can also be evaluated to assess how well they fulfill e.g. security and usability goals, such as by applying the National Institute of Standards and Technology (NIST) Guidelines for Access Control System Evaluation Metrics [35].

2.6 Existing Solutions

Authorization is not a new problem for application developers, and having been studied since the 1970s [82], many solutions have been proposed over the years. They include open-source libraries, built-in framework solutions, enterprise-grade standard solutions, platform-specific proprietary solutions, and paid third-party ACaaS (Access Control-as-a-Service) and SECaaS (Security-as-a-Service) solutions.

This section reviews some of the most commonly used solutions for authorizing web applications. The primary focus is mainly on *policy-as-code* style authorization solutions that express all authorization policies using either a custom DSL, such as *Rego* or *XACML*, or a general-purpose programming language, such as *JavaScript*.

Examples of policies defined using some of the following solutions can be found in Appendix A.

2.6.1 XACML

XACML or the *eXtensible Access Control Markup Language* is an XML-based language for defining policies, standardized by *OASIS*, the Organization for the Advancement of Structured Information Standards [66]. It is primarily designed for expressing ABAC policies, but it can also be extended to suit other access control models [59].

The architecture of XACML, shown in Figure 9, is uncoincidentally similar to that of ABAC as shown in Figure 8. The entry point to XACML is through the *Policy Enforcement Point (PEP)*, which receives an access request and returns a grant or a denial decision. The PEP receives the decision from the *Policy Decision Point (PDP)* which accesses policies through the *Policy Administration Point (PAP)* and all relevant attributes via the *Policy Information Point (PIP)* [66].

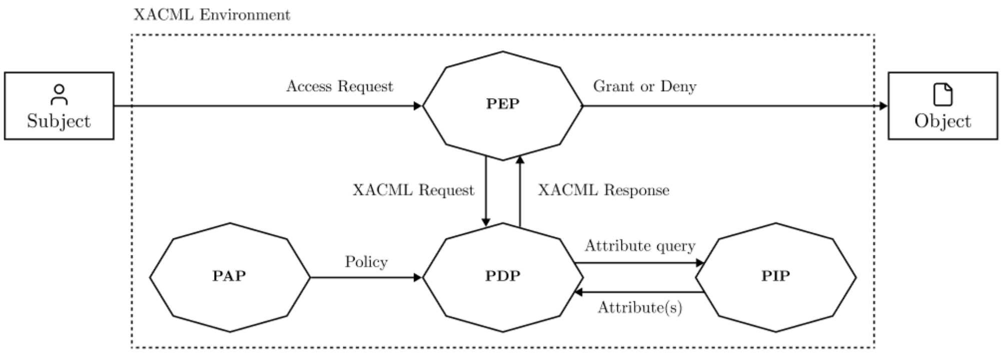

Figure 9: Simplified architecture of XACML [66].

The PAP is primarily interacted with through writing XML-based policies as code. The PIP directly interacts with the data sources such as the database to access relevant attributes.

XACML also supports *obligations* and *advice* in decisions, very similar to the obligations and recommendations of OrBAC [66]. This allows the system to obligate or suggest to perform certain action upon requesting access.

Although standardized, XACML is most likely not suitable for generic web development. Its design is enterprise-oriented [66] and the XML-based policy language is considered heavyweight and verbose for lighter applications. Additionally, it cannot be directly integrated into the web application but it must be run as a separate *sidecar* service, which is interacted with through HTTP.

2.6.2 OpenPolicyAgent (OPA)

OpenPolicyAgent (OPA) is an open-source, general-purpose policy engine that provides an unified method of enforcing policies [68]. It is based on the Rego language, which is a high-level declarative policy language inspired by Datadog [68]. The Rego language allows arbitrary JSON input and output and does not enforce any specific authorization model. Unlike the XML-based XACML, the language is also more concise and testable.

The ecosystem of OPA is focused on authorization in the cloud, e.g. microservices, kubernetes, CI/CD and APIs, and does not have specialized tooling for web development. OPA must be integrated either as a separate service, similarly to XACML, or alternatively by compiling Rego to WebAssembly (WASM) and running the WASM module in your application [67].

2.6.3 OpenFGA and Zanzibar

OpenFGA (Open Fine-Grained Authorization) is an open-source implementation of an authorization system based on the proprietary Google Zanzibar system [71]. While it is able to model any authorization system, it is particularly designed for fine-grained ReBAC authorization systems [71].

The Zanzibar system proposed by Google is designed to support millions of authorization requests per second and powers the ReBAC system of multiple Google products such as YouTube, Cloud and Photos [77]. Zanzibar and OpenFGA are both based on defining relationships between users, resources, and actions using their respective but very similar policy languages and providing a list of *tuples* that instantiate the defined relationships [71]. Their algorithms can then performantly evaluate the rules against the given set of tuples to make authorization decisions.

Google Zanzibar and OpenFGA are mostly very similar, but have several differences, e.g. in their policy languages [70] and usage patterns [72].

OpenFGA is again required to be hosted as a separate service, however unlike XACML or OPA, it provides a dedicated SDK for web developers for integrating and enforcing policies [73].

2.6.4 Cedar

Cedar defines another DSL, the Cedar policy language, for defining policies as code. It is designed to be easy to understand and explicitly designed for authorization, combining RBAC, ABAC and ReBAC aspects [43]. Unlike XACML, Rego or OpenFGA, it is based on a verified formal model and enables analyzability of policies, thus increasing safety [43].

The Cedar language defines the available entities and their attributes and available actions, as well as the permissions for specific subject-action-object tuples [7]. It also provides SDKs for integrating into web applications, however its JavaScript/WASM based implementations have not yet been widely adopted [23, 24].

2.6.5 Auctoritas

Auctoritas, as mentioned in Section 2.5.8, is a system proposed to implement Entity-Based Access Control or EBAC, a combination of ABAC and ReBAC [16, 50]. It has, however, not seen adoption yet [59], as there is no existing implementations of the Auctoritas system available yet, only academic descriptions and proposals.

2.6.6 Other Access Control Languages

The languages listed above do not constitute a comprehensive list. The list of available access control languages is too large to meaningfully cover in this thesis. To give a sense of scale, NIST lists 26 different access control policy languages in their guidelines for evaluating access control systems [35].

2.6.7 Open-Source Libraries

The two most adopted open-source libraries for implementing authorization (that are not SDKs of a proprietary service) are *CASL* [22] and *Casbin* [18, 65].

CASL is the most adopted out of all open-source options [65] and implements a simple ABAC system for securing applications. It is based on defining *abilities* declaratively in JavaScript. Abilities consist of a subject or subjects and an action, defined either as a positive or a negative rule, with the option to specify object attributes which must match and limit the fields which can be accessed [21]. For example:

```
import { defineAbility } from '@casl/ability';

// Builder pattern for defining abilities
function buildAbility(user) {
  return defineAbility((can, cannot) => {
    // User can update their own articles
    can("update", "Article", { authorId: user.id });
    // ... other policies
  });
}

// Example usage
const ability = buildAbility({ id: "1" });
const canAccess = ability.can("update", "Article", {
  authorId: "1",
  title: "Example"
});
```

On the other hand, *Casbin* is an open-source multi-model authorization library for multiple languages [18]. However, its JavaScript version has noticeably less adoption compared to CASL [65]. It mentions support for models such as ACL, RBAC, ABAC, ReBAC, BLP, Biba and more [19]. However, it places a strong emphasis in its feature set and documentation on RBAC and management of roles [20].

Casbin is based on *PML*, the *PERM modeling language*, where PERM stands for *Policy, Effect, Request, Matchers* [44]. Using the PERM model, when a *request (R)* is received, *matchers (M)* are used to determine the *policies (P)* that apply, evaluating which results in an *effect (E)* that either allows or denies the request as a response [44]. This is another example of multi-policy support, which requires a conflict resolution algorithm such as the default *deny-overrides* algorithm by which any denial denies the entire request.

```
[request_definition]
r = sub, obj, act

[policy_effect]
e = some(where (m == true))

[matchers]
m = sub.role == "admin" ||
  (act == "read" && (
    sub.department == obj.department || sub.id == obj.ownerId
  )) ||
  (act == "write" && (
    sub.id == obj.ownerId
  ))
```

Listing 1: Example ABAC-style PML policy.

The implementations of Casbin take the PML policy configuration as an input to create an *enforcer*, which implements the PML parsing, resolution and evaluation algorithm. The enforcer is then used in code to query and enforce the policies using a request-response model [20].

```
const enforcer = await newEnforcer('model.conf', 'policy.csv');
const isAllowed = await enforcer.enforce(sub, obj, act);
```

Listing 2: Example usage of a Casbin enforcer in TypeScript.

There are other libraries attempting to provide authorization solutions, such as `accesscontrol` [1], `cancan` [2], `rbac` [4] or `easy-rbac` [3]. However, they all have marginal adoption, as they are created 9–12 years ago and some are weakly maintained or abandoned [65].

2.6.8 Platform-Specific Solutions

Many *Platform-as-a-Service (PaaS)* providers offer built-in authorization solutions. These include Firebase security rules [32], MongoDB Atlas RBAC authorization [60], Supabase Row-level Security (RLS) policies [87] and Appwrite permissions [9]. These are commonly limited in their flexibility and may only allow for specific access control models. Firebase Rules and Supabase RLS are created to allow safe database access from client-side applications and are custom solutions directly integrated to their respective platform databases [87, 32]. These authorization technologies are often proprietary, cause platform lock-in and solve primarily platform-specific problems. Typically they cannot be directly transferred to other systems not built on the same platform.

2.6.9 Proprietary Third-Party Solutions

The last solutions discussed are proprietary third-party solutions that fall under the category of *ACaaS (Access Control-as-a-Service)*, a subset of Security-as-a-Service (SECaaS) [85]. These include access control specific providers such as Oso Cloud based on the Polar language [75], Permit.io based on the Open Policy Administration Layer (OPAL) with a UI for managing policies [78], Cerbos [25], Aserto [10] or the Zanzibar-based AuthZed [13]. Additionally, many authentication providers such as Auth0 [11], WorkOS [94], Clerk [26] and Kinde [40] provide their own authorization solutions. Some of these, for example Cerbos, provide self-hostable open-source alternatives of their solutions on top of their managed solution.

Most of the proprietary third-party solutions come with trade-offs. The application typically does not own its data and is locked in to the authorization provider. All authorizable requests also require an additional request to the external authorization service. Still, they are often feature-rich, fast to implement, have great tooling and support most use cases and authorization models.

Typically, the integrated access control solutions provided by authentication providers introduce lock-in not only in the authentication layer of the application but also in the authorization layer. They also introduce a high level of coupling between the authentication and authorization layers. Many of them offer only simple RBAC solutions, and a much smaller feature set compared to the third-party providers focused explicitly on authorization.

Integrating these solutions is a classic "buy versus build" decision [12]. This thesis will mainly focus on open-source-based "built" solutions.

3 Kilpi — Designing a Solution

This section introduces *Kilpi*, an open-source TypeScript library for implementing authorization in web applications, as explored in the previous Chapter. This section walks through the requirements, design goals, and architecture of Kilpi.

3.1 Goals and Requirements

Kilpi is designed to work in all TypeScript-based applications, especially web applications. This introduces multiple requirements and design goals to make Kilpi suit a wide range of use cases. Firstly, Kilpi must be a TypeScript-based open-source library distributed via the NPM registry for it to be accessible by all TypeScript and JavaScript developers. The following subsections discuss the other goals and requirements that guided the design of Kilpi in more detail.

3.1.1 Agnostic Design

Although web applications can be built with a wide variety of languages, Kilpi is limited by design to only JavaScript- and TypeScript-based applications. Even in JS/TS-based web applications, there is a wide variety of frameworks and libraries, authentication solutions and authorization requirements. To serve most web applications, Kilpi is designed to be agnostic in multiple ways:

– **Framework and library agnostic**

  Kilpi does not rely on the specific framework or libraries used in the application. It must be usable with any TypeScript-based web framework or library, such as React, Angular, Vue, Svelte, Next.js, Nuxt, Express, or NestJS. Kilpi may however require plugins or adapters to work with specific frameworks or libraries.

– **Authentication agnostic**

  Kilpi must not rely on any specific authentication implementation. It must be able to work with all authentication solutions, from open-source libraries to commercial Authentication-as-a-Service providers. Kilpi requires a *subject adapter* to connect to any authentication implementation to receive the subject.

  Additionally, Kilpi is agnostic towards the type of subject returned from the authentication system. Any entity (full user object, identifier for third-party system or even `null`) is considered a valid subject and provided to the authorization system as is.

– **Authorization model agnostic**

  Kilpi must work with any authorization model and should not impose any specific model (such as RBAC, MAC or OrBAC) on the application. Any authorization model must be able to be implemented with Kilpi, and the application may even implement multiple models if required. This is discussed more in the following section.

3.1.2 Policy-Based Access Control Model

As discussed in Sections 2.4 and 2.5, there is a wide variety of available authorization models that web applications can pick from and hybridize or customize to suit their needs. Kilpi should be designed to suit all these models.

Different models require different relations or attributes such as roles in RBAC [28] or the *Uses* relation between organizations, objects and views in OrBAC [39]. Some attributes and relations are centrally managed, such as in MAC [59], some are user-managed, such as in DAC [6] and some may even be automatically managed by the system such as a ReBAC application that infers the relationships from existing data [29].

Kilpi must be designed such that it neither implements any specific relationships or attributes, or relies on any implementation existing in order to work, nor should it dictate any way of managing the authorization-related entities. This approach ensures that Kilpi is able to work with all authorization models and applications, and it does not impose any system on any application. Kilpi must rather be designed to act as a *framework* on which authorization systems and policies can be built, independent of any application-specific implementations. Therefore, Kilpi is *not* an authorization system by itself, but a tool for building one.

For this reason, the design of Kilpi is based on a variation of a *Policy-Based Access Control (PolBAC)* model as discussed in Section 2.4.7. Kilpi is designed on a simpler version of PolBAC where policies are defined for each action separately. Given an action a, its corresponding $Policy(a)$ is defined as a *policy function* as follows:

$$Policy(a) : (s, o, e) \mapsto Decision$$

such that s, o and e are the subject, object and environment respectively, and the decision is either a grant or a denial. The relationships and attributes of the subject, object and environment are denoted as functions, such as $Roles(s)$, $IsParentOf(s_a, s_b)$, $ParentFolder(o)$ or $Date(e)$.

The model will be referred to as *Functional Policy-Based Access Control (FPBAC)*. This definition enables implementing all previously discussed authorization models. Figures 10 and 11 give examples showing how this model can be applied to implement e.g. RBAC (Figure 10) and EBAC policies (Figure 11). The object and environment can sometimes be omitted, e.g. when the action `files:read` is granted to all subjects with the role of $Reader$. This policy is not dependent on the specific object or environment, only the subject and the action, which implicitly contains the *class* of the object.

This approach maps nicely to a functional implementation in software as well. In practice, the set of all allowed actions A is finite and relatively small and it is practical to define policies per action. Applications are also commonly developed in vertical slices, action by action [79], making it more practical to define policies action by action as well.

The assumed functions ($Roles(s)$, etc.) are easy to implement in code as well, as e.g. object attributes or functions that fetch the data from elsewhere as shown in Figure 3. Any generic policy functions and authorization models can then be defined

$$Policy(\texttt{documents:write}) : (s, o, e) \mapsto \begin{cases} Grant \text{ if} & Editor \in Roles(s), \\ Deny \text{ else.} \end{cases}$$

Figure 10: RBAC policy "documents can be written with the Editor role" expressed in the FPBAC model.

$$Policy(\texttt{posts:view}) : (s, o, e) \mapsto \begin{cases} Grant \text{ if} & Visibility(o) = Public \\ & \lor\ s = Author(o) \\ & \lor\ s \in Friends(Author(o)) \\ Deny \text{ else.} \end{cases}$$

Figure 11: EBAC policy "posts can be viewed by anyone if public, else only by author and friends" expressed in the PolBAC model.

based on this model specific data. This includes both custom policy functions as well functions, which implement the formal rules of models such as OrBAC [39].

```
// Roles(s) := s.roles
type Subject = { id: string, roles: Role[] };

// Friends(user) := fetchFriends(user.id)
function fetchFriends(userId: string) {
  return sql`SELECT user_id FROM friends
             WHERE friend_id = ${userId}`;
}
```

Listing 3: Example implementation of model-specific attributes and relations in code.

Unlike PolBAC, which supports making decisions based on multiple policies simultaneously [97], the FPBAC approach only evaluates a single policy at a time as e.g. $Decision = Policy(a)(s, o, e)$. The application layer may, however, evaluate multiple policies at the policy evaluation point and manually combine their results using any boolean logic, such as $Decision = (Decision_1 \land Decision_2) \lor Decision_3$. This approach simplifies authorization logic by removing ambiguity: a policy is always explicitly defined to return either a grant or a denial, no conflicting decisions can ever occur when evaluating a policy. If evaluating multiple policies, any conflicts must be resolved manually and explicitly. Decision conflict resolution is not a part of the design of Kilpi, whereas policy conflicts are typical in many other systems, such as XACML [83]. Furthermore, this increases security by lowering the risk of broken and misconfigured access control rules [69] caused by ambiguity.

The only authorization model specific implementations in Kilpi must be fully optional utilities for the most common use cases in common authorization models, such as optionally defining a role hierarchy on the authorization system level instead of requiring it to be explicitly implemented in the application or database layers. These are not discussed further as they do not relate to the core issue of designing an authorization framework, and thus are out of scope for this thesis.

3.1.3 Policies as TypeScript Functions

To be an automated authorization system, Kilpi defines policies as code, similarly to other well-established solutions such as XACML [85]. This allows automatic policy evaluation, policy versioning and much more. However, instead of defining a custom DSL for authoring policies, Kilpi leverages capabilities of TypeScript and defines all policies as TypeScript functions.

Kilpi models policies as a policy set, with each action in A mapped to a single policy function with the following signature.

$$\{\ Policy(a) : (Subject, [Object]) \mapsto Decision \mid a \in A\ \}$$

A policy receives the subject whose type is received from the subject adapter, and optionally an object of any type. The object is made optional, as some policies (e.g. a `documents:read` policy that allows all authenticated users) may not need specific information on the object's identity or attributes, only the object's type, which is implicitly provided in the action. For example, `documents:read` implies the object is a document. The action itself is also implicitly provided, as a policy is defined for each action separately. Additionally, as policies are evaluated within the TypeScript runtime, they have automatic access to the environment, such as time and date, as well as any third-party systems.

The policy function returns a decision, which either represents an access grant or denial, with optional additional metadata such as the authorized subject or the reason for denial.

Using TypeScript functions instead of a custom DSL has multiple advantages:

– **Familiarity**

  The developer already knows TypeScript and does not need to learn the syntax and semantics of a new DSL.

– **No separate evaluation runtime or engine**

  As the policies are TypeScript functions, they can simply be evaluated by calling them. There is no need to implement or integrate a separate runtime or engine to parse and evaluate the DSL to receive an authorization decision.

– **Type-safety, inference and autocompletion**

  TypeScript's static type-checking [54] feature allows for a certain level of static analysis when defining and enforcing policies, especially when combined with the type inference system [56]. Additionally, TypeScript is able to improve developer experience by autocompleting [57] policy keys, subject and object attributes and more.

- **Existing tooling**

  From the TypeScript language server [91] and ESLint for TypeScript [90] to more advanced tooling, the developer can access the existing ecosystem of tools that can assist with writing, maintaining and even testing policies.

- **Access to all TypeScript features and runtime**

  TypeScript is a fully featured programming language with APIs to access the environment, network and much more. This ensures that no special APIs for specific scenarios need to be learned by the developers or implemented by the authorization system or DSL.

- **Asynchronous policies**

  The policies may be asynchronous and fetch data from any data source, such as a database, an API or a file system. This enables even more complex policies and dynamic data access patterns. It also eliminates the need for a separate formalized *Policy Information Point (PIP)*, which defines the mechanism for retrieving additional information and attributes required for authorization.

- **Composability**

  TypeScript enables functional programming and treats functions as first-class citizens [36]. This makes it easy to compose policies from smaller, reusable functions, improving modularity and maintainability. All policies must still explicitly return a decision. The policies themselves can not be directly composed. For example the following is not allowed:

$$Policy(a) : (s, o, e) \mapsto \begin{cases} Grant \text{ if} & Policy(b)(s, o, e) = Granted \\ & \wedge\ Policy(c)(s, o, e) = Granted, \\ Deny \text{ else.} & \end{cases}$$

  Instead, the policies can be composed from shared smaller functions, such as:

$$IsAdmin : (s) \mapsto admin \in Roles(s)$$

$$IsEditor : (s, o) \mapsto editor \in Roles(s) \vee Owner(o) = s$$

$$Policy(d) : (s, o, e) \mapsto \begin{cases} Grant \text{ if } IsAdmin(s) \vee IsEditor(s, o), \\ Deny \text{ else.} \end{cases}$$

3.1.4 Server-First Authorization

For an authorization system to be able to be considered secure, all policies must be evaluated and enforced on the server [58, 76]. While policies can be evaluated on the client, those authorization should never be trusted and should only be used to improve the user experience of the application. For this purpose, Kilpi is designed for server-first authorization and enforces and evaluates all policies on the server. This also enables the policies to fetch additional data during evaluation as discussed previously in Section 3.1.3.

Kilpi also provides a secondary client-side mechanism, which is able to fetch authorization decisions from the server. This is primarily intended only for altering the user interface and improving user experience, not for protecting critical resources. While the policy enforcement point is moved to the client using this mechanism, the policy is still evaluated on the server.

3.1.5 Web-Focused Authorization

Kilpi is focused on solving authorization challenges especially for web applications. While Kilpi must be designed as a web-agnostic solution that has APIs for any type of applications, it should also provide additional features and utilities that address web-specific authorization challenges.

The user interface (UI) of a web application must be able to be dynamically altered based on the user's permissions, by e.g. disabling buttons or hiding elements based on the user's permissions. However there is a large number of very different rendering paradigms on the web, for example static HTML, single-page apps (SPAs), server-side rendering (SSR), server-side generation (SSG), hydration, islands, streaming and more with both imperative variants such as jQuery and declarative component-based approaches such as React and Angular [41, 53, 31, 74]. Some of these render UI on the server, some on the client. For this reason, Kilpi must be able to work across a variety of environments and paradigms, on the server and on the client.

Additionally, Kilpi must be designed to work well in the most common authorization situations on the web. This includes protecting actions, mutations, queries, HTTP APIs and endpoints, as well as all pages and routes.

Finally, Kilpi must be able to do this across all JavaScript runtimes, such as Node.js, Deno and Bun [27] as well as on all client environments from the different browser runtimes to Hermes used in React Native mobile applications [52].

3.1.6 Centralized Policy Layer

Kilpi will follow the approach of most existing authorization solutions, which define policies *centrally*, managed separately from the rest of the application code. All policies are defined in a centralized policy set. The policies are only accessed through a Policy Enforcement Point (PEP), which is given the required inputs to evaluate a policy or policies to reach a decision. The decision is then left to the application to be handled.

A *centralized policy layer* offers multiple benefits. Separately defined policies with a PEP to enforce them reduces the duplication of implicit policy definitions spread across different parts of the application. Policies and business logic are separated, and tey can be individually maintained and updated at a single point. This approach solves the problem caused by the cross-cutting nature of authorization. Without centralized policies and a PEP, the application must implement the responsibilities of the policy enforcement, decision, administration and information points in multiple places, leading to a complex, unmaintainable, error-prone and thus unsecure codebase.

3.1.7 Other Goals and Requirements

This section lists other secondary design goals for Kilpi.

- **Extensibility**

  Kilpi should be designed to be extensible with plugins or other extensions to allow complex interactions and integrations, and to account for non-standard authorization requirements, even by third-party developers.

- **Low Integration Cost and Gradual Adoption**

  Kilpi should not require infrastructure or DevOps work to adopt in any new or existing project. It should work using the existing runtime and services unlike sidecar architectures of e.g. XACML [66] or OPA [67]. Kilpi should also be gradually adoptable, allowing an application to implement Kilpi policy by policy with only minor changes to existing code.

- **Developer Experience and Type-Safety**

  As a TypeScript-based open-source library, Kilpi should be designed for the developers. It should offer the best possible developer experience, including good documentation, auto-completion and intuitive but flexible API design. Importantly, Kilpi should also prioritize type-safety and aim to provide a seamless TypeScript development experience, especially by utilizing *type inference* [56] to reduce manual type definition work. Additionally, the coupling introduced to any system by Kilpi should be minimal.

- **Auditability**

  The decisions made by Kilpi should be auditable. By design, all decisions are results of dynamically evaluated functions, not statically analyzable permission sets. Kilpi, as a library, is not able to provide the tools or storage for auditing. Instead, a separate solution for audit logging of decisions must be able to be integrated. The metadata of decisions should also be designed to improve auditability.

- **Performance**

  Kilpi should not act as a performance bottleneck. Only if fetching the subject or additional data for a policy is slow, should the authorization step introduce latency. The Kilpi instance itself should be fast and lightweight with minimal performance overhead and attempt to optimize performance by e.g. caching.

– **Minimal Lock-in**

  Complementary to the goal of low integration cost and gradual adoption, Kilpi should aim to minimize the lock-in of a codebase adopting Kilpi. This is aided by utilizing universal authorization concepts, an explicit API, the open-source nature of the library and the use of TypeScript functions instead of a custom DSL.

3.2 Architecture

Figure 12 shows an overview of the functional architecture of Kilpi. Within the bounds of the application, a request to access a resource or perform an action is made, but intercepted by the Policy Enforcement Point (PEP) similarly to most authorization architectures. The PEP relays the action and objects to the *policy evaluator* (synonymous to policy decision point, PDP, but renamed to clarify intent). Based on the given action, the policy evaluator retrieves the corresponding policy function from the policy set (comparable to the policy administration point, PAP).

The subject is not provided to the policy evaluator. Kilpi requires a *subject adapter*, discussed in more detail in subsequent sections, which retrieves the current subject from the application's authentication provider. Any additional data from other data sources, either internal or external to the application, are also retrieved during the evaluation of the policy function if defined.

Given all the action, policy, subject and object, the policy evaluator then evaluates the policy, which outputs a decision. The decision is returned to the application via the PEP. The application then continues or abandons processing the request based on the authorization decision.

Figure 12 shows the *Kilpi instance* as a component of the application contained within application bounds, not as a separate service. The policy evaluator is fully managed by the Kilpi instance, but the instance exposes the subject adapter, policy enforcement point and policy set to the application.

If the policy evaluator runs expensive calculations, Kilpi could make a system susceptible to denial-of-service (DoS) attacks. To mitigate this, an application should implement rate limiting and other DoS mitigation techniques before the PEP to avoid unnecessary load on the policy evaluator. Rate limiting and DoS protection is, however, out of scope for Kilpi and this thesis.

To allow for even more custom behavior, Figure 13 shows the component architecture of Kilpi. This figure illustrates how Kilpi is extended and interacted with on the client, or used in more advanced ways.

The Kilpi instance exposes *hooks* to react to events emitted by Kilpi and to intercept and customize the behavior of Kilpi. Additionally, the Kilpi instance allows for custom plugins, which have access to the Kilpi instance and its hooks and can modify its behavior, extend the public API of the instance and more.

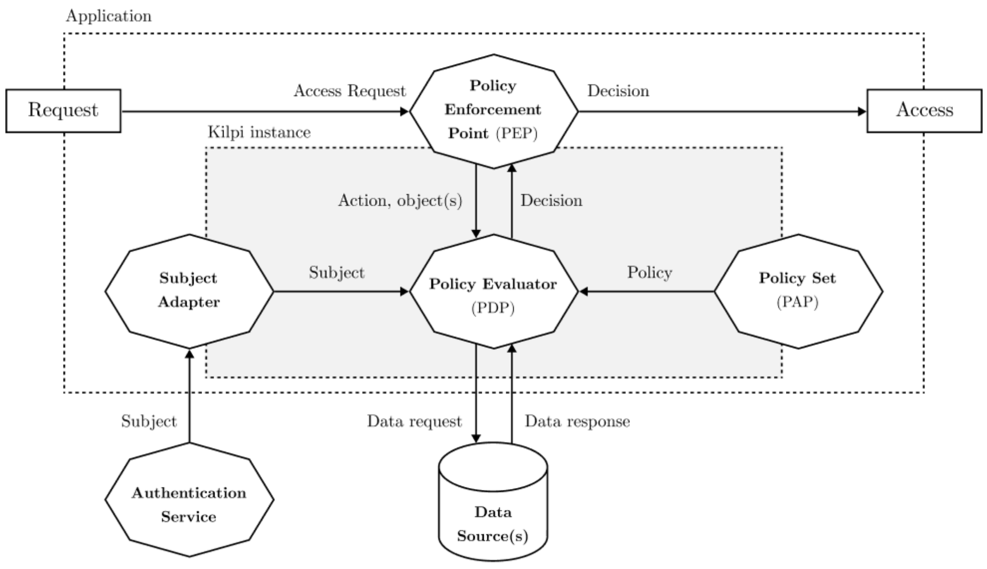

Figure 12: Overview of the functional architecture of Kilpi.

The Kilpi instance and its plugins also have access to the *scope*, which can be thought of as the current request context. It is used to enable request-scoped data and behaviors. For example, after fetching the subject from the subject adapter, the Kilpi instance caches it in the scope. This avoids potentially expensive refetching of the subject during subsequent policy evaluations within the same request. Another example is to define a request-scoped implicit `onUnauthorized` behavior, discussed more in Section 3.4.3 on the *implicit authorization flow*. The scope can be used to store any data or behavior for the duration of a single request, and it can importantly be accessed by plugins.

For client-side authorization, the Kilpi instance can optionally also expose an HTTP API for requesting authorization decisions from. The HTTP API is implemented as a plugin, and it can be called either manually or via the Kilpi client-side SDK.

3.3 Policy Definition Language

As discussed above in Section 3.1.3, Kilpi defines policies as TypeScript functions. The source code files, which define the policy set and all policies, act as the primary policy administration point (PAP).

Policies are defined as functions in a `PolicySet`. The set maps all available actions to their corresponding policy functions. The functions receive the `Subject` and optionally the `Object` and return a `Decision`, and the policy functions can either be synchronous or asynchronous. The example in Listing 4 shows simple policies for the `read` and `write` actions, where reading is always allowed to all authenticated subjects regardless of the object, but writing is only allowed on own objects. The `as const satisfies Policyset<Subject>` typecast clause is required to improve type inference within Kilpi. The rest of the listings in this thesis omit the

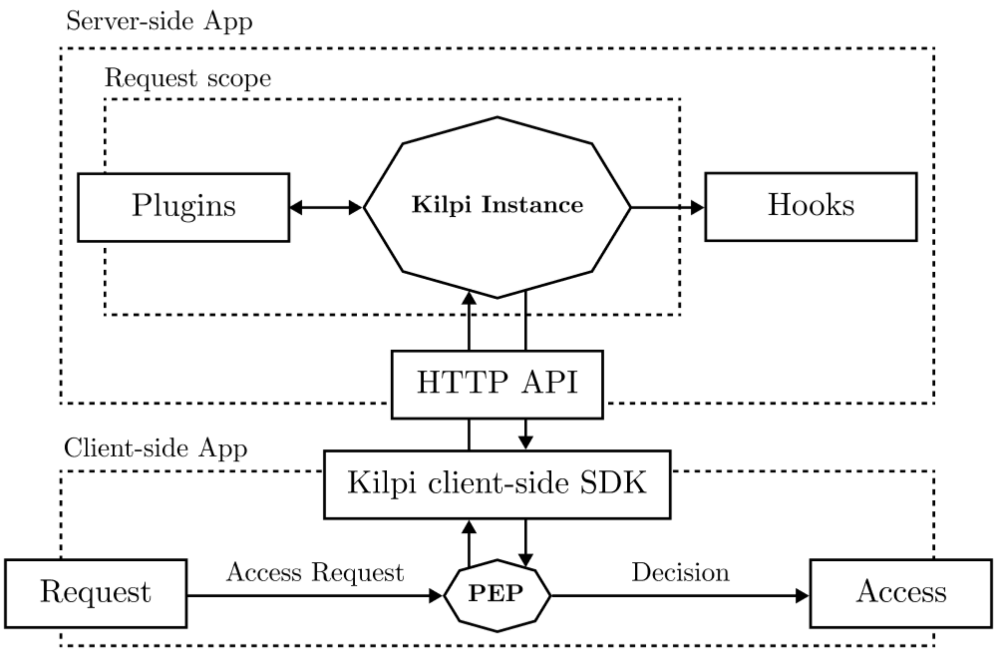

Figure 13: Component architecture of Kilpi.

typecast clause and imports for conciseness.

```
import { deny, grant, type Policyset } from "@kilpi/core";
import type { Subject, Document } from "...";

const policies = {
  read(subject) {
    if (!subject) return deny();
    return grant(subject);
  },
  write(subject, object: Document) {
    if (!subject) return deny();
    if (subject.id !== object.ownerId) return deny();
    return grant(subject);
  }
} as const satisfies Policyset<Subject>;
```

Listing 4: Simple example policies using TS-based policy API of Kilpi.

Kilpi allows nesting and composition of policies, which is useful for e.g. grouping actions by object type as shown in Listing 5. In this case, the *action keys* are automatically generated as e.g. `documents:read` or `comments:write` based on the structure of the `PolicySet`.

As mentioned, policies can also be asynchronous and fetch data from external data

```
const postPolicies = {
  read(subject, doc: Document) { ... },
  write(subject, doc: Document) { ... },
}

const commentPolicies = {
  read(subject, comment: Comment) { ... },
  write(subject, comment: Comment) { ... },
}

const policies = {
  posts: postPolicies,
  comments: commentPolicies,
}
```

Listing 5: Example of nested structured policies.

sources, such as the database as shown in Listing 6.

```
const policies = {
  files: {
    async download(subject, files: File) {
      if (!subject) return deny();
      const folder = await fetchFolder(file.folderId);
      if (!folder.users.includes(subject.id)) return deny();
      return grant(subject);
    }
  }
}
```

Listing 6: Example of an asynchronous data fetching policy.

Instead of simply returning `true` or `false` to signal a granted or a denied decision, Kilpi returns a decision object via `grant(subject)` or `deny()`. Firstly, passing the authorized subject allows Kilpi to narrow the type of the subject [55]. The example in Listing 7 shows how this improves the developer experience (the `Kilpi.authorize` function will be discussed later).

Similarly, the `deny()` function can optionally be provided a `message`, a `type` and `metadata`, all of which allow customizing behavior on denied requests. For example, the `type` can be used to distinguish between users who were unauthorized due to being unauthenticated from those who were on the free plan, and to handle these denied decisions differently (redirecting to the login page or the billing page respectively).

Overall, this allows for flexible authorization strategies, complex logic, function composition and utility functions for defining policies, data fetching, all while utilizing a familiar and powerful language with all of its tooling and benefits.

```
// UserID if signed in, else null
type Subject = string | null;

const policies = {
  read(subject) {
    if (!subject) return deny();
    return grant(subject); // TS knows subject can't be null here
  }
}

// `s` is inferred to be a non-nullable string
// Thus, no `if (!s)` check needed
const s = await Kilpi.authorize("read");
```

Listing 7: Example of type narrowing with granted subject.

It is worth noting that the PAP could also be implemented elsewhere, using e.g. custom application logic or a third-party authorization provider. This moves the logic out of the policy functions, while still using the Kilpi model, and allows e.g. policies to be maintained using a graphical interface aimed towards non-technical managers.

Appendix A provides a comparison of policy definition languages and methods by listing examples of a simple policy implemented using the TypeScript-based policy API of Kilpi as well as multiple other implementations using different approaches and DSLs.

3.4 Integration and Enforcement Patterns

This section provides an overview of the implementation of the policy enforcement point and how it is applied in application code. This section covers the basic principles and patterns required for authorizing all parts of a web application, from instantiation, to the subject adapter to the different methods of enforcing policies in different environments and contexts.

3.4.1 Instantiating Kilpi

Instantiating Kilpi is straightforward using the `createKilpi` function as shown in Listing 8. A Kilpi instance requires two parameters. The `getSubject` function that acts as the subject adapter (discussed in more detail in Section 3.4.2) and the `policies` as discussed in previous sections. Additionally, the Kilpi instance can be provided with `plugins` (Section 3.6) and other `settings`. The resulting `Kilpi` object acts as the Kilpi instance, through which all authorization is performed.

```
import { createKilpi, EndpointPlugin } from "@kilpi/core";

export const Kilpi = createKilpi({
  getSubject,
  policies: { ... },
  // Other configuration (plugins, settings)
});
```

Listing 8: Example of instantiating a Kilpi instance.

3.4.2 Subject Adapter

As discussed in Section 2.2, authentication precedes authorization. This insight allows Kilpi to remove a step from the business logic of the application by removing the call to the authentication service. Instead, Kilpi is provided an adapter, through which Kilpi can call the authentication service automatically.

The subject adapter pattern is implemented using the `getSubject` function. It is a function that optionally receives the current request as an argument and returns any data representing the current subject. Typically this is a user object for authenticated users, or `null` or a guest object for unauthenticated users. It may however also contain any other data or represent any other entity type accessing the application. Listing 9 shows a simple example of this pattern. The adapter pattern enables low coupling between the authentication and authorization layers of the application, while reducing complexity caused by the cross-cutting natures of both authentication and authorization.

```
async function getSubject(request: Request) {
  // Call authentication provider
  const user = await AuthenticationService.getUser(request);

  // Return appropriate subject representation
  if (!user) return null;
  return { id: user.id, name: user.name }
}
```

Listing 9: Example of a simple `getSubject` adapter.

Another aspect how Kilpi reduces repetitive application logic is that the PEP methods (discussed in the next chapters) do not accept the `request` as an argument. Instead, Kilpi typically requires wrapping your request in a *scope*, which is used to automatically route the argument of the `getSubject` function. The scope pattern is discussed in more detail in Section 3.4.9.

Data that is commonly required by policies, related to the caller can optionally be included in the subject object. This includes e.g. the user's role(s), organization memberships or permissions.

For improving performance, when a scope is available, the subject is cached in the scope after the first fetch to avoid multiple, potentially expensive authentication service calls during a single request. This behavior can be opted out of if necessary.

3.4.3 Handling Authorization Decisions

Kilpi exposes several different methods for enforcing authorization. These are introduced in subsequent sections. All methods share a common structure and require the `action` and optionally the `object` to be called. The subject is automatically provided to the policy by Kilpi using the subject adapter.

The methods, however, differ by their behavior and return value and fall into two different categories. The first category is known as the *explicit authorization flow*, which is more traditional and requires manually handling all decisions. On the other hand, the *implicit authorization flow* assumes an `onUnauthorized` handler and automatically throws on unauthorized requests. These approaches can be even used simultaneously. The following subsections review how they can be applied as well as their benefits and drawbacks.

3.4.4 Explicit Authorization Flow

The first way of controlling access is using the more traditional explicit authorization flow. This approach provides full control to the developer, however it also requires always explicitly handling the unauthorized cases. The Kilpi instance exposes two methods for this purpose. The first method is `Kilpi.isAuthorized`, which returns a boolean indicating whether the current subject is authorized to perform the given action on the given object. The application must then handle both the authorized and unauthorized cases.

```
async function deleteDocument(id: string) {
  const doc = await db.documents.getById(id);
  const allow = await Kilpi.isAuthorized("docs:delete", doc);

  if (!allow) throw new HTTPError(403);

  await db.documents.deleteById(id);
}
```

Listing 10: Example of explicit authorization flow using `Kilpi.isAuthorized`.

The `Kilpi.getAuthorizationDecision` method is similar to the previous method above, but it returns the entire decision object with all associated metadata about the decision, which subject access was granted to, or e.g. the reason of the denial. It allows for similar but more complex behavior when compared to a simple boolean, such as handling different types of denials differently.

```
const decision = await Kilpi.getAuthorizationDecision(...);

// Access denied, handle accordingly
if (!decision.granted) {
  if (decision.type === "not-subscribed") {
    throw new Redirection("/subscribe");
  } else if (decision.type === "not-authenticated") {
    throw new Redirection("/login");
  } else {
    throw new HTTPError(403);
  }
}

// Access granted, continue processing...
```

Listing 11: More detailed example of explicit authorization flow using `Kilpi.getAuthorizationDecision`.

3.4.5 Implicit Authorization Flow

Kilpi also provides an alternative approach known as the implicit authorization flow. This approach is designed to automatically throw on unauthorized requests. The throwing is controlled with an `onUnauthorized` handler. This way of handling authorization reduces the common `if (!allow) { ... }` boilerplate in application code, and allows for cleaner code. The flow is implemented using the `Kilpi.authorize` method, as shown in Listing 12.

```
async function deleteDocument(id: string) {
  const doc = await db.documents.getById(id);
  // Throws if unauthorized
  const user = await Kilpi.authorize("docs:delete", doc);
  await db.documents.deleteById(id);
}
```

Listing 12: Example of implicit authorization flow using `Kilpi.authorize`.

The implicit flow is especially useful in scenarios, where multiple authorization checks happen during a single request or when the applications has duplicated behavior for unauthorized requests in multiple places. These are both common in web applications, where the user must be authorized to access the route as well as all data in the route.

The implicit flow assumes that the application should throw an exception on unauthorized. This is the case in many frameworks, that represent e.g. `HTTP 403 Forbidden` errors or redirections as exceptions. The flow is also built on the assumption that the authorization check is a hard stop, and processing the request should immediately stop on an unauthorized request. It also assumes that regardless of the authorization check, the denial behavior should be either global or per-request scoped.

By default, when no `onUnauthorized` handler is available, Kilpi throws an `UnauthorizedError`. The global handler is provided to the Kilpi instance at creation, as shown in Listing 13.

```
export const Kilpi = createKilpi({
  // ...,
  settings: {
    async defaultOnUnauthorized(denial) {
      if (denial.type === "not-authenticated") {
        throw new Redirection("/login");
      }
      throw new HTTPError(403);
    }
  }
})
```

Listing 13: Example of setting up a global `onUnauthorized` handler.

To be even more specific, a request-scoped handler can be provided (stored in the scope, discussed in Section 3.4.9) using the `Kilpi.onUnauthorized` method as given in Listing 14. This is useful for example to protect certain routes of a web application by redirecting any unauthorized requests to a specific page.

```
Kilpi.onUnauthorized(async (denial) => {
  throw new Redirection("/home");
})
```

Listing 14: Example of setting up a request-scoped `onUnauthorized` handler.

Naturally, as the implicit flow relies on throwables, the thrown exceptions need to be properly handled. Some frameworks automatically handle specific thrown exceptions, such as redirections or HTTP errors. In other cases, an error handler must be defined to catch these exceptions and handle them appropriately.

3.4.6 Protecting Queries

The explicit and implicit authorization flows are useful for protecting actions, routes, mutations and commands, or any discrete operations that either succeed or fail. However, most web applications also read and expose data via queries. While queries can be protected using the already provided methods, Kilpi provides an optional mechanism specifically for protecting queries. These are called *protected queries* created using the `Kilpi.query` constructor method.

A well-designed query should aim to be a pure function, without any side-effects. This makes it easier to cache, memoize or deduplicate for performance. Adding an authorization check to a query, however, introduces a side-effect as the same call may fail or succeed depending on the caller.

Moreover, queries should not always simply succeed or fail, but they should sometimes return either empty data or partial data. For example, a user data query should return only public fields such as name and avatar for callers not authorized to the private data of the user. For authorized callers, the query should include all fields. This is known as *redaction*.

The `Kilpi.query` method allows defining a protected query by wrapping a pure data fetching function with a protector function as shown in Figure 14. The protector is given the input and output of the data fetching function as well as the current subject and it can either throw an exception (using the implicit flow) or return the full data, partial data or empty data (using the explicit flow).

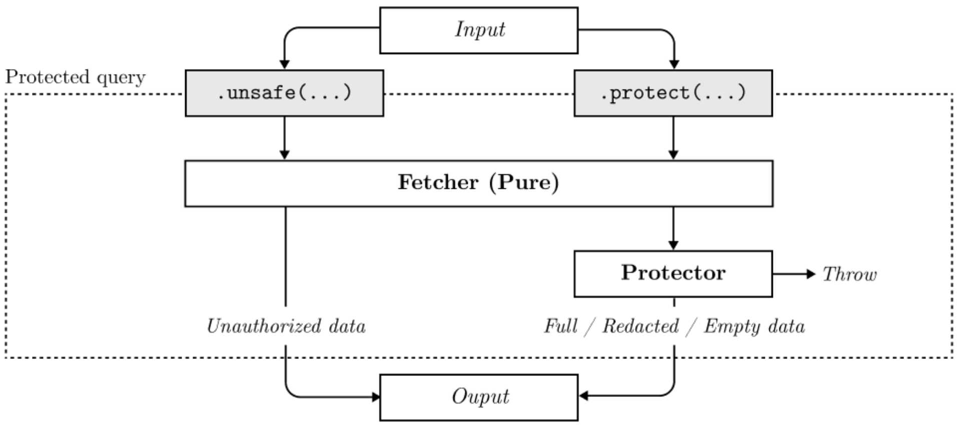

Figure 14: The protected query model of Kilpi.

Additionally, protected queries have two entrypoints, the `protect` and the `unsafe` methods. The `protect` method is the default option, and runs the protector function on the fetched data before returning. The `unsafe` method however allows for calling the data fetching function directly without applying the protector function. This is a useful escape hatch for certain scenarios, primarily those where the queried data will not be exposed to the user.

The query cannot be called directly as a function and must be called via one of the two explicit methods. This is intentional design to enforce the developer to communicate their intent and to avoid accidentally exposing unauthorized data. It should also be much clearer to other developers, that data from `someQuery.protect(...)` is safe to use, whereas data from `someQuery.unsafe(...)` should not be shown to the user.

The example given in Listing 15 shows a simple example query, wrapped with a protector function which implicitly throws an exceptin if the caller is not authorized to read the document. The example in Listing 16 shows a more explicit approach which also redacts the email based on the caller's permissions.

```
// Define a protected query
export const getDocument = Kilpi.query(
  // Provide a pure data fetching function
  async (id: string) => await db.documents.getById(id),
  // Provide the protector implementation
  {
    async protector({ output: doc, input, subject }) {
      // Throw if unauthorized (implicit flow)
      await Kilpi.authorize("documents:read", doc);
      // Return full when authorized
      return doc;
    }
  }
)

// Usage
const doc1 = await getDocument.protect("1");
const doc2 = await getDocument.unsafe("2");
await getDocument("123"); // Error
```

Listing 15: Example of a simple protected query using the implicit flow.

```
export const getUser = Kilpi.query(
  async (id: string) => await db.users.getById(id),
  {
    async protector({ output: user }) {
      const allowed =
        await Kilpi.isAuthorized("users:read-private", user);
      if (!allowed) return { name: user.name, id: user.id };
      return { name: user.name, id: user.id, email: user.email };
    }
  }
)

// Email inferred to be optional (may be redacted)
const user = await getUser.protect("1");
```

Listing 16: Example of a protected query using the explicit flow for data redaction.

Currently, a pre-protector API is also considered in addition to the post-protector API shown above, for optimizing performance by avoiding running queries that are known ahead of time will be unauthorized, regardless of their output. Especially when combined with the implicit flow and an `onUnauthorized`, protected queries make authorizing data access and reads very easy to implement. The example in Listing 17 shows, how little authorization code is required to render a protected web page (the example assumes a React Server Component based framework such as Next.js).

```
async function DocumentPage(props: { id: string }) {
  Kilpi.onUnauthorized(() => redirect("/login"));

  await Kilpi.authorize("routes:documents:view");
  const doc = await getDocument.protect(props.id);
  const comments = await getComments.protect(props.id);

  return (/* UI code */)
}
```

Listing 17: Example of conciseness of the implicit flow in practice.

3.4.7 Client-Side Authorization

The above patterns can be applied to cover most server-side authorization scenarios. However, many applications require authorization decisions to be accessible on the client-side as well. Common uses include dynamically altering the user interface based on the user's permissions, or to avoid making requests that are known to be unauthorized.

For this purpose, Kilpi provides the server-side `EndpointPlugin` to expose the decision endpoint via an HTTP API as well as a client-side SDK to call the endpoint. The primary methods available via the endpoint and the client-side SDK are `KilpiClient.fetchIsAuthorized` to fetch boolean authorization decisions as well as `KilpiClient.fetchSubject` to fetch the current subject. Example usage of the `KilpiClient` is shown in Listing 18.

In codebases or mono-repositories, which co-locate the frontend and backend, the `KilpiClient` instance can also be given the type of the server-side `Kilpi` instance to automatically infer available policies and their types. For performance reasons, the client-side SDK implements request batching, canceling, caching and deduplication to minimize network traffic and latency, as well as utilities to control these behaviors.

3.4.8 User Interface Authorization

Related to client-side authorization but also to server-side authorization, many web applications require the ability to alter their UI based on the user's permissions. Common examples of this are disabling buttons or hiding parts of the page if a user is unauthorized to certain data or actions.

For this reason, Kilpi provides plugins for popular web frameworks to provide authorization functionality directly in the UI. These plugins provide e.g. components to hide or alter parts of the UI as well as hooks to alter component behavior. The components, constructed via the plugin system, are also fully typesafe. Listing 19 shows

```
import type { Kilpi } from "./kilpi.ts";
import { createKilpiClient } from "@kilpi/client";

export const KilpiClient = createKilpiClient({
  infer: {} as typeof Kilpi,
  connect: {
    endpointUrl: process.env.PUBLIC_KILPI_URL,
    secret: process.env.PUBLIC_KILPI_SECRET,
  },
});

const subject = await KilpiClient.fetchSubject();
const canEdit = await KilpiClient.fetchIsAuthorized({
  action: "documents:edit",
  object: { id: "123" },
});
```

Listing 18: Example of instantiating and using the Kilpi client-side SDK.

a client-side example of this using React and the `ReactClientComponentPlugin`, built on top of the client-side Kilpi instance. Similar functionalities are also available for React server components and can be built to any other UI frameworks via the plugin system.

```
// Example of hiding parts of the UI
function UserPage({ user }) {
  return <div>
    <p>{user.name}</p>
    <ClientAccess to="users:edit" on={user}>
      <UserForm user={user} />
    </ClientAccess>
  </div>;
}

// Example of altering component behavior
function DeleteDocumentButton({ id }) {
  const { isAuthorized } = useIsAuthorized("docs:delete", id);
  return <button disabled={!isAuthorized}>Delete</button>
}
```

Listing 19: Example of UI authorization using React components and hooks.

3.4.9 Scope for Request Context

Lastly, this section covers the already often mentioned concept of *scope*. The scope is the term used by Kilpi to refer to the current request context. It works on top of the `AsyncLocalStorage` API [64], which is designed for cross-cutting concerns and aspect oriented programming solutions, such as authorization.

Scope enables using the complete feature set of Kilpi. These features include subject caching, request-scoped `onUnauthorized` handlers as well as certain plugins. Without scope, these features do not work, as there is no scope to which requests can be stored. Storing these values globally would also expose the risk of leaking sensitive data between requests.

Scope can be created in one of two ways. The first method is to wrap the entry point(s) of the application with scope manually using one of the either the `Kilpi.runInScope` or the `Kilpi.scoped` functions. Additionally, the scope can be provided the current request, which receives its type from and is routed as a paremeter to the `getSubject` function. This is shown in Listing 20.

```
// Express.js middleware: Wrap request entrypoint with scope
app.use(async (request, next) => {
  await Kilpi.runInScope(async () => {
    await next();
  }, request);
})

// Next.js endpoint: Wrap API route entrypoint with scope
export const POST = Kilpi.scoped(async (req) => { /* ... */ });
```

Listing 20: Example of manually wrapping an application's entry point(s) with scope.

The other method is to use the `onRequestScope` hook, which is called when Kilpi is resolving the current scope. It allows intercepting the resolving process and sideloading a scope, when the current framework already has request context APIs available. This is, however, most commonly automatically done by plugins, such as the React Server Component plugin which uses the `React.cache` [51] for sideloading and storing the request context.

3.5 Data Access Patterns

Many policies require access to different data to make their decisions. Kilpi provides three primary patterns for accessing data in policies:

1. **Provide as object**

   The object of the authorization function can be any data required by the policy. This data must be *pushed* to the policy at authorization and must be available at the policy enforcement point. This is the primary method of providing non-subject related data to policies. An example of this can be seen in Listing 21.

2. **Provide in subject**

   The subject is not restricted to only communicating the identity of the caller. The `getSubject` function can also access any other data required by the policies, such as the user's role(s), organization memberships or permissions. This is useful, when the data is related to the subject and is commonly required by multiple policies. Listing 22 provides an example of this pattern.

3. **Fetch during policy**

   Any data, which is not pushed as the object or provided in the subject can be fetched during the policy function evaluation, even conditionally. This is useful for data, which is not commonly required by policies, or data which is expensive to fetch and should only be fetched when absolutely necessary. The data can be fetched from any data source, either internal or external to the application. Listing 23 depicts fetching external data inside the policy.

```
// Push object identity and attributes to policy
Kilpi.authorize("docs:read", { id: "d1", ownerId: "u1" });
```

Listing 21: Example of pushing data to the policy as the object.

```
// Provide user attributes in getSubject
async function getSubject() {
const user = await AuthService.getUser();
if (!user) return null;
const roles = await RoleService.getRoles(user.id);
return { id: user.id, roles };
}
```

Listing 22: Example of providing commonly required data in the subjec.

Additionally, as policies are evaluated during the runtime of the application, they have implicit access to the environment. This includes environment variables, request data, time of day and more. This is shown in Listing 24. The three data access methods combined with access to the environment allow access to any attribute or entity required for any authorization model. The appropriate pattern can be selected based on the use case and performance considerations.

3.6 Extensibility and Plugins

As mentioned in the design goals, Kilpi is designed to be extensible. This is achieved primarily through a plugin system. Plugins are prebuilt modules that can easily be added to the Kilpi instance. They have access to the Kilpi instance, its hooks and scope and they can extend the interface of the Kilpi instance to add custom functionality.

```
const documentPolicies = {
  // Allow at most 3 documents on free plan
  async create(subject) {
    if (!subject) return deny(...);
    if (!subject.subscription) {
      const count = await countOwnDocuments(subject.id);
      if (count >= 3) return deny(...);
    }
    return grant(subject);
  }
}
```

Listing 23: Example of fetching data during policy evaluation.

```
const documentPolicies = {
// Allow document edits only during office hours,
// but not when application is under maintenance
edit(subject, doc) {
  if (process.env.STATUS === "maintenance") return deny();
  const hour = new Date().getHours();
  if (hour < 9 || hour > 17) return deny(...);
  return grant(subject);
}
}
```

Listing 24: Example of accessing the environment during policy evaluation.

Plugins are available both for the primary server-side SDK as well as the secondary client-side SDK. They can be both first-party plugins provided by Kilpi itself, third-party plugins provided as separate libraries by independent open-source developers, or custom plugins developed for application-specific purposes.

The current set of first-party plugins includes the following extensions.

– `EndpointPlugin` for exposing the PDP through an HTTP endpoint.

– `AuditPlugin` for integrating a logging or auditing service where authorization decisions can be sent to for auditing purposes.

– `ReactServerComponentPlugin` for easily integrating with React Server Component based frameworks such as Next.js, by providing components and automatic scope.

– `ReactClientComponentPlugin` (Client-side) for easily integrating with React on client-side applications via hooks and components.

Plugins are typesafely installed by passing them to the Kilpi instance at creation, with optional arguments to customize the behavior of the plugin. Listing 25 has an example of integrating the `EndpointPlugin`. A similar plugin system is also available for the client-side SDK, allowing for e.g. client-side UI framework integrations and other extensions.

```
export const Kilpi = createKilpi({
  // ...,
  plugins: [
    EndpointPlugin({ secret: "..." }),
    // More plugins...
  ]
})
```

Listing 25: Example of installing a plugin to the Kilpi instance.

4 Discussion

The previous chapter presented the design of Kilpi as an open-source solution for implementing scalable authorization systems in web applications. This Chapter will take the presented design and discuss it in the context of existing research and solution and evaluates it against the requirements and goals set forth in Section 3.1. Lastly, this chapter discusses the known limitations of Kilpi.

4.1 Validation of Design Goals and Requirements

This first subsection discusses the goals and requirements from Section 3.1 and evaluates how well they are met by Kilpi. The evaluation will be based on experience from implementing Kilpi in multiple production applications.

– **Framework and library agnostic design**

  Kilpi is sufficiently agnostic with respect to the chosen framework and library. The Kilpi documentation already has implementation guides for six frameworks and libraries [61] and it is not limited to those options, but it can easily be extended to even more alternative technologies, without official plugins or installation guides.

  The only framework-specific implementations and assumptions are isolated to separate integration plugins that solve framework or library specific integration challenges, such as for React server components.

– **Authentication agnostic design**

  The subject adapter pattern has proven to be a good solution for decoupling Kilpi from the authentication provider. It works in multiple contexts and has allowed for an easy migration path from one authentication provider to another in a production application, with minimal authorization changes.

– **Policy-based access control model and policies as TypeScript functions**

  Kilpi succesfully implements the FPBAC model using TypeScript-based policies. The usage of TypeScript also achieves all TypeScript benefits listed in Section 3.1.3 (familiarity; no separate evaluation runtime or engine; type-safety, inference, and autocompletion; existing tooling; access to all TypeScript features and runtime; composability).

  The improved developer experience (DX) of the policy language in TypeScript helps to avoid security compromises caused by misunderstood security configurations or either intentional or accidental misconfiguration or omission of any authorization configuration. Additionally, it removes the learning curve of a new DSL. There is also no need for any policy conflict resolution algorithms, and no limitations to what authorization models can be implemented or what data the policies can access, unlike with some access control DSLs.

– **Authorization model agnostic**

  The generic FPBAC model has so far been tested with RBAC, ROBAC, ReBAC and ABAC style policies, and it has proven to be flexible enough to implement all of them. Furthermore, all of these models have succesfully been combined in a single application. This gives confidence in stating that Kilpi is flexible enough to implement most authorization models.

– **Server-first authorization**

  The Kilpi library focuses on server-side authorization. Client-side authorization has been made a secondary concern, separated to its own SDK, integrated using a server side HTTP API plugin. All decisions are by design made on the server. This is also reflected in the documentation. While using the core Kilpi library on the client-side may be possible, it is not explored or documented.

– **Web-focused authorization**

  Kilpi has been tested in multiple web applications, and it has proven to be a viable solution in the context. The generic authorization APIs solve practically every use case. The implicit authorization flow discussed in Section 3.4.5 has dramatically reduced the amount of authorization boilerplate in application code. The authorization components and client-side SDK provided by Kilpi have also proven useful in altering the user interface both on the server and on the client.

– **Centralized policy layer**

  Kilpi achieves its design goal to implement a centralized policy layer by separating all policy definitions into a single composable policy set object. In practice, this has reduced the work required to maintain policies.

– **Extensibility**

  The Kilpi plugin system makes Kilpi extensible by design. The flexibility of the system has been tested by the first four very different first-party plugins, and it has been proven to be satisfactory. Further validation of the extensibility of the system will be possible as more plugins are developed.

– **Low integration cost and gradual adoption**

  Integrating Kilpi into an existing application is easy, however, it involves two slightly complex steps, that could make the learning curve steeper than anticipated. These are the subject adapter and the scope. On top of the intuitive policy enforcement API, these are the only places, where the application is coupled to Kilpi. These are well documented and they must only be set up once.

  After integrating the Kilpi core library, gradual adoption is made possible by defining policies one by one, and replacing existing authorization logic at policy enforcement points with the Kilpi APIs. In summary, Kilpi fulfills this requirement.

– **Developer experience and type-safety**

While this thesis is unable to provide an objective measure of the DX of Kilpi, this part attempts to evaluate it based on subjective experience.

Overall, DX has proven to be good. This is aided by intuitive and explicit API naming and good documentation. The centralized policy layer and unified authorization APIs have also reduced the work required to implement and maintain authorization logic. The previous evaluation items have also proven anecdotal evidence of good DX in production web applications.

The coupling introduced by Kilpi is very low. The agnostic design, centralized and explicit policy definitions, separation of the PEP, PDP and PAP have all contributed to this and are based on established best practices of most existing solutions. The subject adapter also helps reduce coupling to the authentication provider.

DX is greatly improved by Kilpi's type-safety. All primary APIs are strictly typed, maximize inference, and their design is always based on the capabilities of TypeScript. This includes auto-completion of available policies and required attributes at the PEP. The only examples, where types must be manually defined include the following.

  – `as const satisfies Policyset<Subject>` as recommended boilerplate for all policy set objects, if not defined directly in the Kilpi constructor.
  – The type of the object in policies, which must be manually provided.
  – The optional type of the request or context provided to the subject adapter.

Additionally, some systems designed to allow for more complex application specific use cases. For exampke, the metadata of the authorization decision and hooks have looser type-safety to allow usage in more contexts. These however only affect advanced use cases not part of the core use cases of Kilpi.

Other trade-offs in DX include the scope. It introduces copmlexity and may be difficult to integrate in some frameworks. However, it is a powerful tool which enables a much richer feature set and better performance, and has therefore been justified and included in the design of Kilpi. For this reason, it has been also made optional but heavily recommended, including warnings to the developer if the scope is omitted when features, which use the scope are used.

– **Auditability**

Kilpi provides no built-in auditing features, except for the `AuditPlugin`, which is used to collect and flush all authorization decisions to a custom handler. In practice, as Kilpi is an open-source library and not a hosted service, this is the best possible solution. The auditability of Kilpi could only be improved by providing more specialized plugins for specific third-party logging services, instead of only a generic plugin.

– **Performance**

  Compared to solutions that use a sidecar architecture, Kilpi has the advantage of running on the same runtime as the application. This removes all network latency and serialization overhead. In practice, the performance impact of Kilpi is minimal. The only noteworthy latency introduced by Kilpi is the subject adapter, which typically involves a database call or a network call. This would typically be required in any case, and the subject caching mechanism of Kilpi can even help reduce the impact of multiple calls to the authentication provider.

  The only serious performance consideration are slow policy definitions. If the developer writes slow policies with database calls, complex computations or other network calls, they will affect the performance of all parts of the application that enforce those policies. This is, however, not a limitation of Kilpi itself, but an important consideration for the developer when designing the authorization system.

– **Minimal lock-in**

  As Kilpi is gradually adoptible, it can also gradually be migrated out of. Additionally, Kilpi is not mutually exclusive from other authorization solutions. Even a hybrid approach is possible, where Kilpi APIs are used to enforce policies, but the policy definitions utilize another system, such as calling the API of an external ACaaS provider. Migrating libraries, frameworks, authentication providers or even the entire authorization model is possible with Kilpi.

  The only downside of buying into Kilpi is the potential lack of maintenance or support in the future, inherent to all open-source software. With this in mind, Kilpi can be considered to fulfill the no lock-in requirement.

Generally, Kilpi fulfills its goals and requirements well and is a viable solution in practice.

4.2 Evaluation Against the NIST Guidelines

This section evaluates Kilpi against the National Institute of Standards and Technology (NIST) guidelines for access control systems [35]. The guidelines were designed for evaluating full access control systems, however, Kilpi is a framework for building those systems, not a system itself. For this reason, the evaluation will focus on the types of access control systems that can be built with Kilpi, especially focusing on any potential limitations that Kilpi might impose on the resulting access control systems.

The NIST guidelines are in line with other evaluation criteria found in literature, such as those given by Sifou et al. [86] and Sahafizadeh and Parsa [80]. The following subsections evaluate Kilpi against the four categories of the NIST guidelines, providing a summary of the evaluation but not a full tabular analysis of each guideline separately.

4.2.1 Administration Properties

The first category of the NIST guidelines explores the administrative properties of access control systems. Kilpi provides decision auditing capabilities (although a separate auditing service is required) and appropriate syntactic and semantic support for specifying AC rules via the TypeScript language. It spans the all parts of the application and can be used to authorize anything, however it is designed to primarily run on a single host. The HTTP API plugin can be used to slightly overcome this limitation. Kilpi is easy to configure into TypeScript-based systems and provides good policy management capabilities, but only for developers.

Kilpi does not support multi-policies, impact analysis, runtime policy changes or privilege and capability queries. Privilege and capability queries are discussed in Section 4.3.6. Overall, while Kilpi has several minor limitations in the administrative category, it performs well in most important aspects.

4.2.2 Enforcement Properties

The enforcement properties category discusses the types of policies that can be enforced and how they are enforced.

Firstly, Kilpi supports policy combination and composition via TypeScript function composition. Bypassing policies is possible if the PEP itself is omitted or bypassed. As stated previously, Kilpi does not support multi-policies and neither conflict resolution nor prevention is therefore necessary.

Kilpi can be made fully operationally and situationally aware, as it has runtime access to any properties possible. By design, Kilpi offers very granular control and can express any policy or model using the TypeScript-based FPBAC approach.

By design, Kilpi does not have a mechanism to enforce the principles of least privilege or separation of duties, but it allows building a system that can support them. Similarly, Kilpi itself is not inherently *safe*, but the system built on top of it can be. Overall, Kilpi excels in this category, and any system built on top of it can be made very secure and expressive.

4.2.3 Performance Properties

Kilpi adds no performance overhead compared to inline authorization logic and is integrated with the authentication function via the subject adapter. As policies are TypeScript functions, policy retrieval is practically immediate. In summary, the baseline performance of KILPI is excellent, unless the developer introduces bottlenecks with slow-running policy functions or a slow subject adapter.

4.2.4 Support Properties

Finally, the last category of the NIST guidelines is support properties. These are mostly out of scope for Kilpi and not applicable. There are no policy import or export systems, user interfaces or access control management APIs.

Policies can, however, be verified, as the Kilpi API is testable via TypeScript testing tools, if tests are written. As policies are part of the TypeScript source code, they are also version controlled by default. In this regard, Kilpi does not impose any limitations, neither does it provide any additional support tooling.

4.3 Limitations

Some limitations were already discussed in previous evaluation sections. This section summarizes the known limitations of Kilpi as an authorization system. Some of these limitations may be addressed in future development, whereas some limitations may be by design and inherent to the chosen architecture.

4.3.1 Policy Management

In addition to managing policies as code by developers, Kilpi does not provide any built-in functionality for policy management. All access control management intended for non-developers must be implemented separately as an application specific feature, integrated into the defined policies. Alternatively this can be achieved by integrating an ACaaS solution in tandem with Kilpi. As policy management is a highly application and domain specific isue, Kilpi, by design, does not attempt to solve it.

4.3.2 Naming Policies

Using Kilpi instead of inline policies (simple `if` statements) requires naming each policy and action that can be performed in the application. This may worsen DX and add more mental load, as the developer is required to not only think of good names for actions, but a good structure and naming scheme to keep the policies consistent, organized and useful. This is by design, as the explicitness of policies makes authorization systems more scalable and maintainable, as well as understandable by other developers.

4.3.3 Limited Support for Authorization Model Specific Features

Kilpi by design does not attempt to implement any specific authorization model. However, it means that Kilpi can be a worse solution than a dedicated solution for a specific authorization model, as Kilpi has limited support for the features and requirements of any given model.

Especially ReBAC may be difficult to implement performantly, as it may require complex graph queries in policies. The Google Zanzibar system [77] will most likely outperform most self-built solutions in this regard. This is, however, another justified trade-off, as the ReBAC needs of many applications are not as complex as those requiring Zanzibar. Most applications requiring ReBAC can often be satisfied with simpler solutions, such as good database and query design.

Future development for Kilpi may however include built-in utilities and tooling for some of the most common authorization models, such as RBAC, ROBAC and

ABAC. These will primarily consist of only optional utilities. The Kilpi documentation also includes guides for implementing the most common models, with performance considerations.

By design, Kilpi also does not provide any GUI or API for managing application permissions, roles, or other concepts, as all of these are application specific and left to the developer.

4.3.4 Simplified Implementation of Policy-Based Access Control

The FPBAC model implemented by Kilpi is a simplified version of the general policy-based access control (PolBAC) model discussed in Section 2.4.7, and introduces some limitations by design [97].

The FPBAC model does not support implicit multi-policies. As Kilpi maps all actions to a single policy function, given a subject-object-action tuple at the PEP, only a single policy is evaluated. This means that complex policies must be implemented as a single policy function, instead of multiple smaller policies. On the other hand, this simplifies the model and avoids the need for implicit conflict resolution algorithms.

Additionally, there is no built-in hierarchy of policies. Given two policies, `read` and `write`, there is no way to define that `read` is always allowed if `write` is allowed. Instead, both policies must be defined separately. The policies can, however, share implementations using function composition, as they are based on TypeScript.

4.3.5 TypeScript and JavaScript Only

By nature, Kilpi is currently limited to TypeScript and JavaScript applications. This is a trade-off for performance and DX, as many applications and businesses fit this requirement. The primary audience, for whom this is a serious limitation, are enterprise businesses using multiple technologies, and requiring a unified, enterprise-oriented authorization solution across all applications, such as XACML [66]. This is not the intended audience of Kilpi.

Kilpi can expose its decision endpoint using a plugin. This endpoint can be called not only from the client but from any service or application. This way, especially with SDKs for other languages, Kilpi could be used as an authorization server for any application. This is, however, not the intended use case of Kilpi, and no SDKs for other languages exist yet.

4.3.6 Lack of Policy Queries by Subject or Resource

A feature of many simpler authorization systems, especially PerBAC systems based on the existence (or non-existence) of specific permission objects, is the ability to query the system for e.g. all actions that a subject can perform, all objects a subject can access, or all subjects that can access a specific object.

As Kilpi defines policies as arbitrary functions, this is not possible in Kilpi, unless explicitly evaluating each policy for each subject-object pair. This functionality, if required, must be implemented separately by the developer, in an application specific manner. Though, this is not a requirement for most web applications.

4.3.7 Risks of Open-Source Software

Kilpi is an open-source library. This has many benefits, but it also introduces some risks inherent to open-source software. Primarily, the main concern is the lack of dedicated support and maintenance, limited support, potential security vulnerabilities, as well as lack of maintenance or abandonment.

These are, however, challenges faced by all open-source software. Kilpi should aim to mitigate these risks with good documentation, active maintenance and a healthy open-source community, as well as by being open to contributions and sponsorships.

5 Conclusion

This thesis has presented Kilpi, an open-source library for implementing authorization in TypeScript and JavaScript based applications, especially web applications. Kilpi is a flexible library built in TypeScript, able to implement most authorization models and systems. It is designed to be easy to use and integrate into existing applications, while also being performant and scalable.

Additionally, this thesis has proposed the *Functional Policy-Based Access Control (FPBAC) model* — a simplified version of the general policy-based access control model — which is implemented by Kilpi. The model maps each action to a single dynamically evaluated policy function, which is given the subject and object as parameters and which has access to any required attributes, entities or environment data during evaluation.

Kilpi has been implemented and released as an open-source library, available on the NPM registry [62] and documented at `https://kilpi.vercel.app`. Kilpi is currently under active maintenance and is already used in multiple production applications. The current version of the Kilpi core library is `0.19.0` and changes from this thesis may occur in future versions.

5.1 Summary of Results

According to the evaluation of Kilpi against the goals and requirements set out in Section 3.1, Kilpi fulfills its primary objective as a flexible authorization library for JS/TS applications well. It was found to perform well in various use cases and environments using different access control models and authentication providers. The FPBAC model at the core of Kilpi has proven to be a simple yet powerful way to express authorization logic, and the plugin system has enabled easy extensibility and customization of Kilpi for different needs. Evaluating Kilpi against the NIST guidelines and other literature showed proof that Kilpi suits the generic requirements of authorization systems very well.

Future development includes more testing on the viability of Kilpi in different use cases. Most of future development will be maintenance and improving the developer experience, documentation and usability of Kilpi, e.g. by reviewing the design of the current implementation of scope. Most future development is in essence minor, as the core of the Kilpi library already solves the main problems of authorization in web applications well.

5.2 Future Work

There remain many possible directions in which future research on this topic could be taken.

Work on evaluating Kilpi is still required. The scope of this thesis did not include a comprehensive and objective evaluation of Kilpi, and this remains an important area for future work. Both quantitative and qualitative research on Kilpi, especially in comparison to other solutions, both open and closed source, is still required. This may include performance benchmarks, usability studies, case studies, and more. Additionally, Kilpi may also be tested as a solution in contexts outside of web applications, such as desktop applications, mobile applications, and more.

Future work also includes any extensions of Kilpi to solve more problems related to authorization in general, or in specific access control nichés. These include new plugins, any future development of the core library, new tools and libraries built around Kilpi, and more. Additionally, the idea and design of Kilpi can be ported to other languages and ecosystems other than the TS/JS ecosystem. This is possible due to Kilpi being published as open-source.

References

[1] "accesscontrol," npm package, accessed 29.8.2025. [Online]. Available: https://www.npmjs.com/package/accesscontrol

[2] "cancan," npm package, accessed 29.8.2025. [Online]. Available: https://www.npmjs.com/package/cancan

[3] "easy-rbac," npm package, accessed 29.8.2025. [Online]. Available: https://www.npmjs.com/package/easy-rbac

[4] "rbac," npm package, accessed 29.8.2025. [Online]. Available: https://www.npmjs.com/package/rbac

[5] M. G. Al-Obeidallah, D. G. Al-Fraihat, A. M. Khasawneh, A. M. Saleh, and H. Addous, "Empirical investigation of the impact of the adapter design pattern on software maintainability," in *2021 International Conference on Information Technology (ICIT)*. IEEE, 2021, pp. 206–211.

[6] K. Albulayhi, A. Abuhussein, F. Alsubaei, and F. T. Sheldon, "Fine-grained access control in the era of cloud computing: An analytical review," in *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2020, pp. 0748–0755.

[7] Amazon Web Services, "Cedar policy language documentation," accessed 29.8.2025. [Online]. Available: https://docs.cedarpolicy.com/

[8] Angular, "Best practices: Security," website, 2025, accessed 15.8.2025. [Online]. Available: https://angular.dev/best-practices/security#

[9] Appwrite, "Permissions," website, 2025, accessed 2.9.2025. [Online]. Available: https://appwrite.io/docs/advanced/platform/permissions

[10] Aserto, "Fine-grained, scalable authorization in minutes," website, 2025, accessed 2.9.2025. [Online]. Available: https://www.aserto.com/

[11] Auth0, "Fine-grained authorization that scales with you," website, 2025, accessed 14.8.2025. [Online]. Available: https://auth0.com/fine-grained-authorization

[12] AuthZed, "Openai securely connects enterprise knowledge with chatgpt by using authzed," website, 2025, accessed 2.9.2025. [Online]. Available: https://authzed.com/customers/openai

[13] ——, "Stop building authz start building value," website, 2025, accessed 2.9.2025. [Online]. Available: https://authzed.com/

[14] Z. Babar and K. Bassett, "Multi-tenant design practices for cloud-based solutions," PwC Canada, June 2023, accessed 14.8.2025. [Online]. Available: https://www.pwc.com/ca/en/services/consulting/technology/cloud-engineering

/cloud-technology-insights/multi-tenant-design-practices-for-cloud-based-solutions.html

[15] D. E. Bell and L. J. La Padula, "Secure computer system: Unified exposition and multics interpretation," MITRE Corp, Bedford, MA, Technical Report MTR-2997-REV-1, ESD-TR-75-306, 1976, dTIC Accession Number: ADA023588. [Online]. Available: https://apps.dtic.mil/sti/tr/pdf/ADA023588.pdf

[16] J. Bogaerts, M. Decat, B. Lagaisse, and W. Joosen, "Entity-based access control: supporting more expressive access control policies," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC '15. New York, NY, USA: ACM, 2015, pp. 291–300. [Online]. Available: https://doi.org/10.1145/2818000.2818009

[17] D. Brossard, "Is XACML actually used and implemented?" February 2018, accessed 29.8.2025. [Online]. Available: https://softwareengineering.stackexchange.com/questions/365843/is-xacml-actually-used-and-implemented

[18] Casbin, "casbin," npm package, accessed 29.8.2025. [Online]. Available: https://www.npmjs.com/package/casbin

[19] ——, "Casbin - an authorization library that supports access control models like acl, rbac, abac," accessed 29.8.2025. [Online]. Available: https://casbin.org/

[20] ——, "Casbin documentation - overview," accessed 29.8.2025. [Online]. Available: https://casbin.org/docs/overview

[21] CASL, "Casl documentation - introduction," accessed 29.8.2025. [Online]. Available: https://casl.js.org/v6/en/guide/intro

[22] ——, "@casl/ability," npm package, accessed 29.8.2025. [Online]. Available: https://www.npmjs.com/package/@casl/ability

[23] Cedar Policy, "@cedar-policy/cedar-authorization," npm package, 2025, accessed 2.9.2025. [Online]. Available: https://www.npmjs.com/package/@cedar-policy/cedar-authorization

[24] ——, "@cedar-policy/cedar-wasm," npm package, 2025, accessed 2.9.2025. [Online]. Available: https://www.npmjs.com/package/@cedar-policy/cedar-wasm

[25] Cerbos, "Fine-grained access control in days not months," website, 2025, accessed 2.9.2025. [Online]. Available: https://www.cerbos.dev/

[26] Clerk, "Authorize users," website, 2025, accessed 2.9.2025. [Online]. Available: https://clerk.com/docs/guides/authorization-checks

[27] errilaz, "Awesome js runtimes," GitHub repository, 2025, accessed 26.8.2025. [Online]. Available: https://github.com/errilaz/awesome-js-runtimes

[28] D. F. Ferraiolo and D. R. Kuhn, "Role-based access controls," in *Proceedings of the 15th National Computer Security Conference*, Baltimore, USA, October 1992, pp. 554–563.

[29] P. W. Fong, "Relationship-based access control: protection model and policy language," in *Proceedings of the First ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '11. New York, NY, USA: ACM, 2011, pp. 191–202. [Online]. Available: https://doi.org/10.1145/1943513.1943539

[30] F. Giunchiglia, R. Zhang, and B. Crispo, "Relbac: Relation based access control," in *2008 Fourth International Conference on Semantics, Knowledge and Grid*. IEEE, 2008, pp. 3–11.

[31] Google, "Angular - the web development framework for building modern apps," website, 2025, accessed 26.8.2025. [Online]. Available: https://angular.dev/

[32] Google Firebase, "Firebase security rules," website, 2025, accessed 14.8.2025. [Online]. Available: https://firebase.google.com/docs/rules

[33] Google Trends, "Access control vs authorization: Search interest comparison," website, 2025, accessed 15.8.2025. [Online]. Available: https://trends.google.com/trends/explore?cat=31&date=today%205-y&q=access%20control,authorization&hl=en-GB

[34] V. C. Hu, D. R. Kuhn, D. F. Ferraiolo, and J. Voas, "Attribute-based access control," *Computer*, vol. 48, no. 2, pp. 85–88, 2015.

[35] V. C. Hu and K. Scarfone, *Guidelines for access control system evaluation metrics*. US Department of Commerce, National Institute of Standards and Technology, 2012.

[36] R. H. Jansen, *Hands-On Functional Programming with TypeScript: Explore functional and reactive programming to create robust and testable TypeScript applications*. Packt Publishing Ltd, 2019.

[37] Y. Jiang, C. Lin, H. Yin, and Z. Tan, "Security analysis of mandatory access control model," in *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No.04CH37583)*, vol. 6, 2004, pp. 5013–5018 vol.6.

[38] J. Jurjens, M. Lehrhuber, and G. Wimmel, "Model-based design and analysis of permission-based security," in *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*. IEEE, 2005, pp. 224–233.

[39] A. Kalam, R. Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miege, C. Saurel, and G. Trouessin, "Organization based access control," in *Proceedings POLICY 2003. IEEE 4th International Workshop on Policies for Distributed Systems and Networks*. IEEE, 2003, pp. 120–131.

[40] Kinde, "Access management: The secure approach to containing risk," website, 2025, accessed 2.9.2025. [Online]. Available: https://kinde.com/access-management/

[41] R. Kokam, "10 rendering patterns for web apps," DEV Community, 2025, accessed 26.8.2025. [Online]. Available: https://dev.to/riteshkokam/10-rendering-patterns-for-web-apps-gp4

[42] D. Li, C. Liu, and B. Liu, "H-rbac: a hierarchical access control model for saas systems," *International Journal of Modern Education and Computer Science*, vol. 3, no. 5, p. 47, 2011.

[43] J. Lovelock, "Access management customer use of cedar policy & verified permissions," Recording of a talk at *AWS re:Inforce 2024*, available on YouTube, 2024, https://www.youtube.com/watch?v=vDLI9w9Z-R8, Accessed 29.8.2025.

[44] Y. Luo, Q. Shen, and Z. Wu, "Pml: An interpreter-based access control policy language for web services," *arXiv preprint arXiv:1903.09756*, 2019, accessed 29.8.2025.

[45] Z. Mahmood, Ed., *Continued Rise of the Cloud: Advances and Trends in Cloud Computing*. Springer, 2014.

[46] A. Majumder, S. Namasudra, and S. Nath, "Chapter 2," in *Continued Rise of the Cloud: Advances and Trends in Cloud Computing*, Z. Mahmood, Ed. Springer, 2014, ch. 2, pp. 23–54.

[47] M. Mammass, "An access control model based on the concepts of organization and service for large infrastructures," *Procedia Computer Science*, vol. 148, pp. 571–579, 2019, The Second International Conference on Intelligent Computing in Data Sciences, ICDS2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050919300304

[48] M. Mammass and F. Ghadi, "Access control models: State of the art and comparative study," in *2014 Second World Conference on Complex Systems (WCCS)*. IEEE, 2014, pp. 431–435.

[49] M. Marin, L. Moonen, and A. van Deursen, "A classification of crosscutting concerns," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, IEEE. IEEE, 2005, pp. 673–676.

[50] L. T. Martın, F. O. F. Pena, and A. A. Leiva, "Auctoritas: A semantic web-based tool for authority control," in *XIV Congreso Internacional de Información Info'2016*, 2013.

[51] Meta Platforms, Inc., "cache," React Documentation, 2025, accessed 5.9.2025. [Online]. Available: https://react.dev/reference/react/cache

[52] ——, “Javascript environment,” React Native Documentation, 2025, accessed 26.8.2025. [Online]. Available: https://reactnative.dev/docs/javascript-environment

[53] ——, “React - the library for web and native user interfaces,” website, 2025, accessed 26.8.2025. [Online]. Available: https://react.dev/

[54] Microsoft Corporation, “The basics,” TypeScript Documentation, 2025, accessed 26.8.2025. [Online]. Available: https://www.typescriptlang.org/docs/handbook/2/basic-types.html

[55] ——, “Narrowing,” TypeScript Documentation, 2025, accessed 3.9.2025. [Online]. Available: https://www.typescriptlang.org/docs/handbook/2/narrowing.html

[56] ——, “Type inference,” TypeScript Documentation, 2025, accessed 26.8.2025. [Online]. Available: https://www.typescriptlang.org/docs/handbook/type-inference.html

[57] ——, “Typescript for javascript programmers,” TypeScript Documentation, 2025, accessed 26.8.2025. [Online]. Available: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html

[58] MITRE Corporation, “Cwe-602: Client-side enforcement of server-side security,” Common Weakness Enumeration, 2025, accessed 26.8.2025. [Online]. Available: https://cwe.mitre.org/data/definitions/602.html

[59] A. K. Y. S. Mohamed, D. Auer, D. Hofer, and J. Küng, “A systematic literature review for authorization and access control: definitions, strategies and models,” *International Journal of Web Information Systems*, vol. 18, no. 2-3, pp. 156–180, 08 2022. [Online]. Available: https://doi.org/10.1108/IJWIS-04-2022-0077

[60] MongoDB Inc., “Guidance for atlas authorization,” website, 2025, accessed 2.9.2025. [Online]. Available: https://www.mongodb.com/docs/atlas/architecture/current/auth/authorization/

[61] J. Nevavuori, “Kilpi,” website, 2025, accessed 14.8.2025. [Online]. Available: https://kilpi.vercel.app/

[62] ——, “@kilpi/core,” npm package, 2025, accessed 14.8.2025. [Online]. Available: https://www.npmjs.com/package/@kilpi/core

[63] Next.js, “Authentication: Authorization,” website, 2025, accessed 15.8.2025. [Online]. Available: https://nextjs.org/docs/app/guides/authentication#authorization

[64] Node.js Project, “Asynclocalstorage (async context api),” Node.js Documentation, 2025, accessed 5.9.2025. [Online]. Available: https://nodejs.org/api/async_context.html

[65] NpmTrends, "@casl/ability vs accesscontrol vs cancan vs casbin vs easy-rbac vs rbac vs xacml," website, accessed 29.8.2025. [Online]. Available: https://npmtrends.com/@casl/ability-vs-accesscontrol-vs-cancan-vs-casbin-vs-easy-rbac-vs-rbac-vs-xacml

[66] OASIS, "extensible access control markup language (xacml) version 3.0," OASIS, OASIS Standard, January 2013, accessed 29.8.2025. [Online]. Available: https://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html

[67] Open Policy Agent, "Integrating OPA," accessed 2.9.2025. [Online]. Available: https://www.openpolicyagent.org/docs/integration

[68] ——, "Open policy agent documentation," accessed 29.8.2025. [Online]. Available: https://www.openpolicyagent.org/docs

[69] Open Web Application Security Project Foundation, "OWASP top ten 2021," website, 2021, accessed 13.8.2025. [Online]. Available: https://owasp.org/www-project-top-ten/

[70] OpenFGA, "Configuration language," accessed 26.9.2025. [Online]. Available: https://openfga.dev/docs/configuration-language

[71] ——, "Openfga documentation," accessed 29.8.2025. [Online]. Available: https://openfga.dev/docs/fga

[72] ——, "When to use openfga as the 'source of truth' for authorization data," accessed 26.9.2025. [Online]. Available: https://openfga.dev/docs/best-practices/source-of-truth

[73] ——, "@openfga/sdk," npm package, 2025, accessed 2.9.2025. [Online]. Available: https://www.npmjs.com/package/@openfga/sdk

[74] OpenJS Foundation, "jquery - write less, do more," website, 2025, accessed 26.8.2025. [Online]. Available: https://jquery.com/

[75] Oso, "What is oso cloud?" website, 2025, accessed 2.9.2025. [Online]. Available: https://www.osohq.com/docs/what-is-oso-cloud

[76] OWASP, "Transaction authorization cheat sheet," OWASP Cheat Sheet Series, 2025, accessed 26.8.2025. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Transaction_Authorization_Cheat_Sheet.html

[77] R. Pang, R. Caceres, M. Burrows, Z. Chen, P. Dave, N. Germer, A. Golynski, K. Graney, N. Kang, L. Kissner, J. L. Korn, A. Parmar, C. D. Richards, and M. Wang, "Zanzibar: Google's consistent, global authorization system," in *2019 USENIX Annual Technical Conference (USENIX ATC '19)*. Renton, WA: USENIX Association, 2019.

[78] Permit.io, “Permissions for the ai era,” website, 2025, accessed 2.9.2025. [Online]. Available: https://www.permit.io/

[79] I. M. Ratner and J. Harvey, “Vertical slicing: Smaller is better,” in *2011 Agile Conference*. IEEE, 2011, pp. 240–245.

[80] E. Sahafizadeh and S. Parsa, “Survey on access control models,” in *2010 2nd International Conference on Future Computer and Communication*, vol. 1. IEEE, 2010, pp. V1–1–V1–3.

[81] R. Sandhu, “Rationale for the rbac96 family of access control models,” in *Proceedings of the first ACM Workshop on Role-based access control*, 1996.

[82] R. Sandhu and P. Samarati, “Access control: principle and practice,” *IEEE Communications Magazine*, vol. 32, no. 9, pp. 40–48, 1994.

[83] I. Shamoon, Q. Rajpoot, and A. Shibli, “Policy conflict management using xacml,” in *2012 8th International Conference on Computing and Networking Technology (INC, ICCIS and ICMIC)*, 2012, pp. 287–291.

[84] H.-b. Shen and F. Hong, “An attribute-based access control model for web services,” in *2006 Seventh International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT’06)*. IEEE, 2006, pp. 74–79.

[85] M. A. Shibli, R. Masood, U. Habiba, A. Kanwal, Y. Ghazi, and R. Mumtaz, “Access control as a service in cloud: Challenges, impact and strategies,” in *Continued Rise of the Cloud: Advances and Trends in Cloud Computing*, Z. Mahmood, Ed. Springer, 2014, ch. 3, pp. 55–102.

[86] F. Sifou, A. Kartit, and A. Hammouch, “Different access control mechanisms for data security in cloud computing,” in *Proceedings of the 2017 International Conference on Cloud and Big Data Computing*, ser. ICCBDC ’17. New York, NY, USA: ACM, 2017, pp. 40–44. [Online]. Available: https://doi.org/10.1145/3141128.3141133

[87] Supabase Inc., “Row level security,” website, 2025, accessed 2.9.2025. [Online]. Available: https://supabase.com/docs/guides/database/postgres/row-level-security

[88] B. Tang, Q. Li, and R. Sandhu, “A multi-tenant rbac model for collaborative cloud services,” in *2013 Eleventh Annual Conference on Privacy, Security and Trust*, 2013, pp. 229–238.

[89] R. K. Thomas, “Team-based access control (tmac) a primitive for applying role-based access controls in collaborative environments,” in *Proceedings of the second ACM workshop on Role-based access control*. ACM, 1997, pp. 13–19.

[90] TypeScript ESLint, “Typescript eslint,” website, 2025, accessed 26.8.2025. [Online]. Available: https://typescript-eslint.io/

[91] TypeScript Language Server Team, “Typescript language server,” GitHub repository, 2025, accessed 26.8.2025. [Online]. Available: https://github.com/typescript-language-server/typescript-language-server

[92] J. Vepsäläinen, A. Hellas, and P. Vuorimaa, “The state of disappearing frameworks in 2023,” 2023, approved for WEBIST 2023. [Online]. Available: https://arxiv.org/abs/2309.04188

[93] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser, “Terminology for policy-based management,” Internet Engineering Task Force (IETF), RFC 3198, November 2001, accessed 27.8.2025. [Online]. Available: https://datatracker.ietf.org/doc/rfc3198/

[94] WorkOS, “Role-based access control,” website, 2025, accessed 2.9.2025. [Online]. Available: https://workos.com/rbac

[95] E. Yuan and J. Tong, “Attributed based access control (abac) for web services,” in *IEEE International Conference on Web Services (ICWS’05)*, 2005, p. 569.

[96] Z. Zhang, X. Zhang, and R. Sandhu, “Robac: Scalable role and organization based access control models,” in *2006 International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2006, pp. 1–9.

[97] L. Zhi, W. Jing, C. Xiao-su, and J. Lian-xing, “Research on policy-based access control model,” in *2009 International Conference on Networks Security, Wireless Communications and Trusted Computing*, vol. 2, 2009, pp. 164–167.

A Implementations of Policies in Different Languages

This appendix includes a simple policy that combines role-based access control (RBAC) and attribute-based access control (ABAC) implemented in different access control languages and libraries.

The policy states that a subject is able to read a document, if the subject has the role `reader` (RBAC) and the department of the subject matches the department of the document or always if the subject owns the document (ABAC).

The example is implemented in the Kilpi TypeScript-based functional model, as well as using XACML, Open Policy Agent (OPA) Rego, OpenFGA, Cedar and Casbin languages and the JS-based CASL open-source library.

A.1 Kilpi (TypeScript)

```typescript
import { deny, grant, Policyset } from "@kilpi/core";
import type { Document, Subject } from "./types";

export const policies = {
  documents: {
    async read(subject, document: Document) {
      if (!subject) return deny();

      if (subject.id === document.ownerId || (
        subject.roles.includes("reader") &&
        subject.department === document.department
      )) {
        return grant(subject);
      }

      return deny();
    }
  }
} as const satisfies Policyset<Subject>;
```

A.2 XACML

```
<Policy
  PolicyId="read-document-policy"
  RuleCombiningAlgId=
    "urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides"
  >
  <Target>
    <AnyOf>
      <AllOf>
        <Match
          MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal"
        >
          <AttributeValue
            DataType="http://www.w3.org/2001/XMLSchema#string"
          >
          read
          </AttributeValue>
          <AttributeDesignator
            AttributeId="action"
            Category="urn:oasis:names:tc:xacml:3.0:attribute-category:action"
            DataType="http://www.w3.org/2001/XMLSchema#string"
            MustBePresent="false"
          />
        </Match>
      </AllOf>
    </AnyOf>
  </Target>
  <Rule
    RuleId="owner-permit"
    Effect="Permit"
  >
    <Condition>
      <Apply
        FunctionId=
          "urn:oasis:names:tc:xacml:1.0:function:string-equal"
      >
        <AttributeDesignator
          AttributeId="subject:id"
          Category=
            "urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"
          DataType=
            "http://www.w3.org/2001/XMLSchema#string"
          />
        <AttributeDesignator
          AttributeId="document:ownerId"
          Category=
            "urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
          DataType=
            "http://www.w3.org/2001/XMLSchema#string"
        />
      </Apply>
    </Condition>
  </Rule>
  <Rule
```

```
    RuleId="reader-department-permit"
    Effect="Permit"
  >
    <Condition>
      <Apply
        FunctionId=
          "urn:oasis:names:tc:xacml:1.0:function:and"
      >
        <Apply
          FunctionId=
            "urn:oasis:names:tc:xacml:1.0:function:string-is-in"
          >
          <AttributeValue
            DataType="http://www.w3.org/2001/XMLSchema#string"
          >
            reader
          </AttributeValue>
          <AttributeDesignator
            AttributeId="subject:roles"
            Category=
              "urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"
            DataType=
              "http://www.w3.org/2001/XMLSchema#string"
            />
        </Apply>
        <Apply
          FunctionId=
            "urn:oasis:names:tc:xacml:1.0:function:string-equal"
        >
          <AttributeDesignator
            AttributeId="subject:department"
            Category=
              "urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"
            DataType=
              "http://www.w3.org/2001/XMLSchema#string"
          />
          <AttributeDesignator
            AttributeId="document:department"
            Category=
              "urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
            DataType=
              "http://www.w3.org/2001/XMLSchema#string"
          />
        </Apply>
      </Apply>
    </Condition>
  </Rule>
  <Rule
    RuleId="deny"
    Effect="Deny"
  />
</Policy>
```

A.3 OPA Rego

```
package document.access

default allow = false

allow {
  input.action == "read"
  input.subject.id == input.document.ownerId
}

allow {
  input.action == "read"
  "reader" in input.subject.roles
  input.subject.department == input.document.department
}
```

A.4 OpenFGA

```
type
  user
type
  document

relation
  owner: user
relation
  reader: user

permission
  read = owner or (reader and department_match)

relation
  department_match: user

// Tuples (example data):
// document:doc1, owner, user:alice
// document:doc1, reader, user:bob
// document:doc1, department_match, user:bob
//   To model department matching, you would set department_match
//   for users whose department matches the document's department.
```

A.5 Cedar

```
permit(
  principal,
  action == "read",
  resource
)
when {
  principal.hasRole("reader") && principal.department == resource.department
    || principal == resource.owner
};
```

A.6 CASL (JavaScript)

```
import { AbilityBuilder, Ability } from '@casl/ability';

function defineAbilitiesFor(subject, document) {
  const { can, cannot, build } = new AbilityBuilder(Ability);

  if (subject.id === document.ownerId) {
    can('read', 'Document');
  } else if (
    subject.roles.includes('reader') &&
    subject.department === document.department
  ) {
    can('read', 'Document');
  } else {
    cannot('read', 'Document');
  }

  return build();
}

// Usage example:
const ability = defineAbilitiesFor(subject, document);
const isAllowed = ability.can('read', 'Document');
```

A.7 Casbin with JavaScript

```
// model.conf
[request_definition]
r = sub, obj, act, ownerId, objDept, subDept, roles

[policy_definition]
p = sub, obj, act, ownerId, objDept, subDept, roles, eft

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = (r.sub == r.ownerId && r.act == "read") || (
  r.act == "read" &&
  r.roles.indexOf("reader") != -1 &&
  r.subDept == r.objDept
)
```

```javascript
// JavaScript usage
const { newEnforcer } = require('casbin');
const enforcer = await newEnforcer('model.conf');
const isAllowed = await enforcer.enforce(
  subject.id,
  document.id,
  "read",
  document.ownerId,
  document.department,
  subject.department,
  subject.roles,
);
```