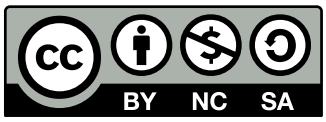
Aalto University
School of Science

Master's Programme in Computer, Communication and Information Sciences

Usable Access Control in Cloud Management Systems

Lucas Käldström

Master's Thesis
2025



Aalto University
School of Science

Author Lucas Käldström

Title Usable Access Control in Cloud Management Systems

Degree programme Computer, Communication and Information Sciences

Major Computer Science – Algorithms, Logic and Computation

Supervisor Dr. Sanna Suoranta

Advisor Dr. Stefan Schimanski

Collaborative partner Upbound

Date 26 May 2025 **Number of pages** 78+5 **Language** English

Abstract

Kubernetes has emerged as a primary method and unified platform for managing server infrastructure, across both public and private clouds. Kubernetes provides a uniform, generic, extensible and declarative application programming interface (API), that both allow humans to self-service, and other platforms to build on top.

The Open Worldwide Application Security Project (OWASP) highlighted Broken Access Control as the top API security weakness in 2023. Kubernetes provides powerful access control mechanisms, but for the authorization and admission request stages separately. There is no concept of conditional authorization, which would bind these two separate stages together uniformly.

This thesis proposes empowering policy authors to write conditionally authorized policies, and a method to implement this, without changing Kubernetes core. This capability makes it easier to write right-sized policies, without having to understand the request stage separation internals.

This thesis also proposes an encoding of Kubernetes API surface into a general-purpose authorization language, Cedar Policy. Cedar was chosen after extensive evaluation of the authorization engine landscape, because it is expressive, safe, fast, and even analyzable in satisfiability modulo theories (SMT) logic. The SMT encoding empowers features such as autocompletion, policy validation, and partially ordering the policies, which all then by the proposed integration can benefit Kubernetes policy authors.

This thesis finds the outlook positive for Kubernetes being able to map into the Cedar data model, and thus be able to take advantage of advanced general-purpose features. This composable approach reduces the engineering effort required in Kubernetes when implementing features. It is found that it is possible to empower policy authors to write uniform, conditionally authorized policies without exposing the Kubernetes-internal two-stage split to the author. However, not all Cedar features required for this Kubernetes integration are yet available as stable features.

Future work includes gathering policy author feedback from the proposed experience, evaluation of making conditional authorization a core primitive of Kubernetes and stabilizing the Cedar features required by Kubernetes.

Keywords Kubernetes, access control, Cedar, satisfiability modulo theories

Aalto-universitetet
Högskolan för
teknikvetenskaper

Författare Lucas Käldström

Titel Användarvänlig åtkomstkontroll i molnhanteringssystem

Utbildningsprogram Computer, Communication and Information Sciences

Huvudämne Computer Science – Algorithms, Logic and Computation

Övervakare Dr. Sanna Suoranta

Handledare Dr. Stefan Schimanski

Samarbetspartner Upbound

Datum 26.5.2025 **Sidantal** 78+5 **Språk** engelska

Sammandrag

Under de senaste åren, har Kubernetes etablerats som en primär metod för att hantera serverinfrastruktur. Kubernetes tillhandahåller användaren en enhetlig plattform för att köra servermjukvara både i publika och privata moln. Detta är möjligt tack vare Kubernetes API, som är enhetligt, generiskt, flexibelt samt deklarativt.

Organisationen The Open Worldwide Application Security Project (OWASP) noterade 2023 att generellt sett är den största svagheten i olika API:n trasig åtkomstkontroll. Kubernetes tillhandahåller kvalitativa åtkomstkontrollfunktioner för sitt API. Tyvärr, är användargränssnittet för dessa funktioner inte enhetligt. Det finns inget koncept för villkorlig auktorisation, en funktion som kunde möjliggöra ett enhetligt användargränssnitt för åtkomstkontrollen.

Denna avhandling föreslår en mekanism för villkorlig auktorisation i Kubernetes och en metod för att implementera detta utan att ändra Kubernetes källkod. Mekanismen gör det lättare för administratörer att skriva en enhetlig och lämpligt omfattande policy, utan att behöva förstå Kubernetes på djupet.

I avhandlingen föreslås också en kodning av åtkomstkontrollen för Kubernetes API till Cedar Policy, ett universellt språk för användarbehörigheter. Cedar är en universell, uttrycksfull, snabb, säker samt analyserbar lösning för åtkomstkontroll. Cedar är analyserbar tack vare att dess policyer kan reduceras till satisfierbarhet modulo teorier (SMT) logik. Denna kodning möjliggör funktioner som automatisk komplettering, validering av policyer samt möjlighet att jämföra policyers storlek.

Avhandlingen konstaterar att det är genomförbart att reducera en stor del av Kubernetes åtkomstkontroll till Cedars datamodell. Därmed kan Kubernetes dra nytta av Cedars avancerade och universella funktioner, vilket minskar utvecklingsinsatsen. Det konstateras att det är möjligt för administratörer att via avhandlingens projekt enkelt skriva enhetliga samt villkorliga policyer. Däremot är inte alla nödvändiga funktioner stabiliserade i Cedar ännu.

Framtida forskningsförslag inkluderar insamling av respons från Kubernetes-administratörer gällande effektiviteten av dessa förbättringar, utvärdering av ifall Kubernetes kunde stöda villkorlig auktorisation som en inbyggd funktion samt stabilisering av de funktioner i Cedar som Kubernetes-integrationen kräver.

Nyckelord Kubernetes, åtkomstkontroll, Cedar, satisfierbarhet modulo teorier

Preface

I want to thank my supervisor Dr. Sanna Suoranta and my advisor Dr. Stefan Schimanski for their guidance.

I want to thank my partner for supporting me throughout the process.

Thank you to Upbound for being the collaborative partner for this Master's thesis.

Thank you to the Cedar and Kubernetes communities for productive discussions about this problem space, in particular Emina Torlak, Darin McAdams and Micah Hausler. Thank you to Jimmy Zelinskie and Micah Hausler for co-authoring talks with me.

Otaniemi, 26 May 2025

Lucas Käldström

Contents

Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
AWS	Amazon Web Services
ABAC	Attribute-Based Access Control
CLI	Command Line Interface
CNCF	Cloud Native Computing Foundation
CRD	CustomResourceDefinition
CA	Certificate Authority
CNF	Conjunction Normal Form
CEL	Common Expression Language
DNF	Disjunction Normal Form
DPoP	Demonstrating Proof of Possession
DAG	Directed Acyclic Graph
DAC	Discretionary Access Control
HTTP	Hypertext Transfer Protocol
HR	Human Resources
ID	Identifier
JSON	JavaScript Object Notation
JWT	JSON Web Token
MAC	Mandatory Access Control
MLS	Multi-level Security
mTLS	Mutual Transport Layer Security
NGAC	Next Generation Access Control
OIDC	OpenID Connect
OWASP	Open Worldwide Application Security Project
PDP	Policy Decision Point
RBAC	Role-Based Access Control
ReBAC	Relation-Based Access Control
REST	Representational State Transfer
SMT	Satisfiability Modulo Theories
SAT	The boolean satisfiability problem
SSO	Single Sign-on
SPIFFE	Secure Production Identity Framework For Everyone
SQL	Structured Query Language
SAR	SubjectAccessReview
SIG	Special Interest Group
TLS	Transport Layer Security
UID	Unique Identifier
VS Code	Visual Studio Code

1 Introduction

Cloud computing [1], the notion of an organization getting access to a virtually unlimited pool of server compute capacity through a self-service interface, has transformed the way organizations operate. More data can now be processed than ever before, regardless of if the organization has chosen to rent the cloud hardware from a public cloud[1], or chosen to build a private data center. Kubernetes [2] has emerged as the common workload orchestration platform, which through its Application Programming Interface (API) allows its users to manage workloads in a self-service manner.

As an increasing amount of data is digitally processed, used, and exchanged, there is an expanding need for suitable access control mechanisms. This applies to both the data itself, and the *metadata* management of the hardware and software where the data is stored, processed, and utilized. The Open Worldwide Application Security Project (OWASP) identified *Broken Object Level Authorization* as the top API security weakness in 2023 [3]. Therefore, this thesis focuses on how to improve the access control primitives for the metadata management API for the infrastructure around the data. The focus is on Kubernetes access control, due to its popularity and influence on the workloads orchestration market. Kubernetes has millions of users, and many cloud providers offer it as a service [4].

Kubernetes API is externally audited to be secure [5], and it provides powerful access control mechanisms already. However, there is always room for improvement. First, there are feature requests that have been open for more than eight years, for example [6]. Second, in recent years, an increasing amount of other non-workload-oriented platforms are built on top of Kubernetes, such as Crossplane [7]. Platforms built on top of Kubernetes do not inherit all the access control mechanisms available to Kubernetes itself. Finally, a lot of engineering effort is needed in order to develop and maintain access control features. Today, Kubernetes has implemented its role-based and node access control itself. However, previously when policy mechanism needed to evolve, Kubernetes chose to use an existing expression language, Common Expression Language (CEL) [8], instead of inventing something custom.

This thesis project has:

1. Identified areas in Kubernetes access control that could be improved.

2. Searched for possible solutions correspondingly. Just like Kubernetes settled on CEL expressions for general policy, it would be good to find out if there are existing authorization-focused projects that integrate well. If there are, Kubernetes would not necessarily have to take on all of the burden of maintaining a more complex access control layer.

3. Implemented novel solutions for providing policy authors with a uniform API across Kubernetes authorization and admission layers.

[1] Such as Amazon Web Services (AWS), Microsoft Azure, or Google Cloud

1.1 Research Questions and Contributions

The research questions are as follows:

1. **RQ1**: How can Kubernetes' API surface, including extension APIs, be modelled in a general purpose, expressive, fast, safe and analyzable access control language?

2. **RQ2**: How can an encoding of Kubernetes API surface in a general purpose authorization language aid in solving some complex and long unresolved feature requests?

3. **RQ3**: How can the user experience of authoring access control policies for Kubernetes API objects be improved and unified?

This thesis contributes to the community as follows. Firstly, a novel encoding of Kubernetes' API surface into Cedar [9] is proposed. It differs from previous encodings [10] in that it abstracts away Kubernetes internal implementation details into one uniform paradigm. This was done in order to improve the policy writing experience for the policy author. Secondly, the policy author can through the uniform Cedar encoding express a *conditional authorization* intent in a straightforward way. The thesis project takes care of translating the policy author's conditional authorization intent into a way Kubernetes understands, without changing Kubernetes. Finally, two Cedar contributions [11], [12] have been proposed and accepted, and two conference talks relating to the thesis topics have been presented in Salt Lake City [13] and London [14] conferences around Kubernetes.

1.2 Methods

The thesis uses various methods:

1. Code review, I research Kubernetes access control primitives, at the level of reading most of the relevant code to get a full picture of the features and limitations.

2. I gathered information by reading user-reported issues around access control expressiveness and usability in the Kubernetes issue tracker[2], and related projects. I engage with the cloud native community and asked users and contributors directly at the following six conferences:

   (a) Cloud Native Rejekts[3] in Salt Lake City, November 10, 2024

   (b) Kubernetes Contributor Summit[4] in Salt Lake City, November 11, 2024

   (c) KubeCon[5] in Salt Lake City, November 12-15, 2024

[2]Available at: https://github.com/kubernetes/kubernetes/issues/
[3]More information at: https://cloud-native.rejekts.io/
[4]More information at: https://www.kubernetes.dev/events/2024/kcsna/
[5]More information at: https://kubecon.io

(d) Cloud Native Rejekts[3] in London, March 30, 2025

(e) Maintainer Summit[6] in London, March 31, 2025

(f) KubeCon[5] in London, April 1-4, 2025

3. Literature and community review for a general purpose, open source[7] authorization engine, whose data model and implementation could support Kubernetes existing access control features and go beyond.

4. I collaborate with the authorization engine community, and contribute to it, if there are features missing, or improvements to make.

5. I reimplement Kubernetes existing access control using the authorization engine and Kubernetes' extensibility mechanisms. The thesis project, the integration which enables this, should support Kubernetes existing access control features, and solve problems identified at step 2.

6. Evaluate if the added complexity of the new features is justified by the new capabilities or easier workflow for the users.

1.3 Structure of the Thesis

Chapter 2 covers general background information relating to access control, prior research, and defines terminology used in this thesis. Chapter 3 covers background related to Kubernetes, and how Kubernetes access control is implemented. Chapter 4 introduces Cedar [9], and how the access control problem can be encoded into logic. Chapter 5 covers the thesis project implementation, and contributions proposed by it. Chapter 6 evaluates the solution in the light of the research questions, community context, and alternative solutions. Chapter 7 summarizes and concludes the thesis, and proposes future work. Appendix A contains more detailed examples that would otherwise disturb the flow is inlined in the main chapters.

[6]More information at: https://events.linuxfoundation.org/kubecon-cloudnativecon-europe/features-add-ons/maintainer-summit/

[7]Preferably, the project should be part of the Cloud Native Computing Foundation, like Kubernetes, so the immaterial property between the dependent and dependency share the same owner.

2 Access Control

The access control features of a system must always be holistically designed and integrated. For example, it is critical to make sure the different logical functions that contribute to the system's security as a whole agree on the semantics of exchanged data. Usually, in order to introduce a new security feature to the system, many different parts of the system need to carefully be changed in tandem to ensure continued system function. Special care need to be taken to make sure the if the system fails, it does so in a controlled and safe way.

A *trust domain* is a perimeter of trust which shares security practices and governance. A trust domain can be defined to "[act] in its own capacity, under its own authority, and is administratively isolated from systems residing in other trust domains." [15]. The trust domain is usually the source of the well-known semantic definitions for the safety-critical information shared within and between domains.

A typical organization divides their software infrastructure into multiple trust domains. The *least privilege principle* recommends that principals should operate with a minimal set of privileges. Partitioning a large system A into smaller trust domains B and C makes it easier to ensure that a principal belonging to only B cannot get any privileges in C.

The defense-in-depth best practice recommends use many layers of security controls at once. This practice ensures that even if one layer is breached, another security control will reject the unauthorized request. One example of this is to reject a request for sensitive information as early as possible, if it is known that there is no chance of it becoming authorized later. This also saves system resources.

2.1 Authentication

Authentication is the process of identifying a user of a software system. The user presents some type of credential, identity proof, to the system in order to prove their identity. In this thesis, the user subject to authentication is referred to as a *principal*, in order to suggest that the principal can either be a human or a *workload* (another software system). The part of the system that performs the authentication process is referred to as an *authenticator*. A trust domain might make use of multiple authenticators, for different sub-parts of the domain.

An *issuer* is a service which keeps track of what principals exist, and issues credentials for principals to present to authenticators. An issuer might provide a mechanism to distribute the issued credentials to the principal. The issuer and authenticator might be centralized and bundled into the same service, or they might be independent services.

The use-case for separating the issuer and authenticator from each other is *principal federation*. Principal federation allows a principal to manage fewer logical identities, by re-using their one identity when authenticating to multiple different systems. For humans logging in to systems, *single sign-on* (SSO) is the canonical example, where the human can log into almost any website with their Google or Microsoft account. However, in a federated setup, the issuer must provide the authenticator with

a mechanism to verify an issued credential. An authenticator might trust multiple issuers.

There are two types of credentials: symmetric and asymmetric. Both credential types are usually associated with a given principal's identity. A symmetric credential is a secret which is shared between anyone needing to either use or authenticate the associated identity. In other words, the symmetric credential is known at least the principal, issuer, and possibly authenticators. Any holder of a symmetric credential can use the identity associated with it. This commonly means that the authenticator can impersonate the principal, which is not a desired feature. Asymmetric credentials do not have this weakness, and are thus considered more secure. With an asymmetric credential, the principal can prove their identity without exposing the credential to the authenticator.

During the authentication process, the authenticator should pay attention to the following [16]:

- *Temporal accuracy*: When a valid credential contains a given property, it is known that this was true when the credential was issued. However, the authenticator should make its own judgement of whether it is likely that the property still holds. For example, if usernames in a system are mutable, `username=lucas` in a credential issued six months ago might be different to the current user with `username=lucas`.

- *Scope and influence*: The authenticator should understand what the intended audience for this credential properties are, and how far the influence should reach. For example: just because a principal has `role=admin` in trust domain A, it does not mean that they have that role in trust domain B.

- *Interpretation and meaning*: Are the semantics of the credential data shared between the issuer and authenticator, or do they diverge? Here common out-of-band administrative processes are useful between trust domain, for example, organizational operating standards.

- *Veracity*: Can the credential data be believed to conform with truth? For example, did the issuer actually mean to sign this data, or was it tricked into doing so? For example, how much trust does an authenticator in trust domain A put in an issuer in trust domain B?

All of this matters, because the second most impactful risk to API security, according to the OWASP Foundation [3], is broken authentication. Credentials should always be time-bound, to reduce the damage of credential theft, and other similar attacks. However, how to perform credential rotation in more detail is out of scope for this thesis.

This section covers the for server systems two most common authentication credentials and corresponding authentication mechanisms: Mutual Transport Layer Security (mTLS) and OpenID Connect (OIDC). Finally, there is a brief introduction to a project called SPIFFE, which provides powerful principal federation mechanisms for client certificates and OIDC.

2.1.1 Mutual TLS

Virtually all internet security is today built on top of the Transport Layer Security cryptographic protocol. In order for a client to securely browse to a website, the website server needs to provide the client with a X.509 server certificate [17] for the website domain. In addition, the client needs to trust the certificate authority (CA) which signed the server certificate. If these two requirements are met, the client has verified the authenticity of the website. However, for normal web browsing, the website usually does not authenticate the client using the same method.

A common and secure authentication mechanism, suitable for server systems, is that the client provides a signed client certificate[8] to the server. This works well for humans using a program or Command Line Interface (CLI) tool to contact the server, or when the client is an automated system. It does not work well in practice, however, when a human sends requests to the server through a web browser. For this use-case, OpenID Connect is a more suitable alternative.

That both the server and client authenticate each other's certificates is called mutual TLS [18]. An X.509 certificate is an asymmetric credential, which consists of a public part (the certificate) and a private part (the key). The certificate is digitally signed by the certificate authority (CA), which acts as the issuer for the principal. A CA can also delegate trust and issuing capabilities to a sub-CA.

Principal federation is by design fairly achieved by when using client certificates. The main requirement is to securely distribute the CA certificate from the issuer to the authenticator. The CA certificate is used by the authenticator to verify the authenticity of client certificates. Practically, the harder part is for the issuer and authenticator to agree on the semantics of the exchanged information about the principal.

The principal must never share the private key with anyone else, because anyone who has the private key can act as the principal. This means that the principal must be in an environment where persistent and secure storage of the key is possible.

If the private key would be leaked to untrusted parties, the issuer can publish a Certificate Revocation List (CRL) [19] to inform authenticators not to trust the leaked certificate. However, if an authenticator is configured to require reading the CRL, and it is unavailable, it must make a choice between availability and consistency [20].

2.1.2 OpenID Connect

OpenID Connect (OIDC) [21] is a collection of standards for authenticating principals that present signed a JSON Web Token (JWT) [22]. OIDC builds on top of TLS and the OAuth 2.0 authorization framework [23]. In this thesis, the terminology of introduced in the beginning of the chapter will be used, as opposed to the terms used in the specification. OpenID Connect is the most popular way to implement principal federation for browser-based services, known as single-sign on (SSO). The OpenID Connect Core 1.0 [24] specification defines the following at the highest level:

1. An identity token[9] format, encoded as a JWT, and semantic meaning of well-

[8] In this thesis, *certificate* always refers to a X.509 certificate.

[9] Referred to as an *ID token* in the specification, but *OIDC token* in this thesis, for clarity.

known token attributes. The OIDC token is a symmetric credential.

2. A browser-based login flow for a principal to complete, in order to acquire a signed OIDC token from the issuer.

3. A framework for authenticators to apply in order to verify the identity of the principal presenting an OIDC token credential. There are in particular three safety-critical properties to verify:

   (a) That the OIDC token is digitally signed by (a for the authenticator) trusted issuer. The authenticator might download the material needed for verifying token authenticity from the issuer's discovery endpoint [24].

   (b) That the OIDC token is meant to be presented to, and thus scoped for, the given authenticator. This is called the *audience* of the token, and it is intended to remedy token forwarding attacks. It is recommended that the token specifies exactly one audience [25].

   (c) That the OIDC token is not expired. Due to the token being symmetric and hard to revoke, it is recommended to keep token validity duration as short as possible. The OIDC protocol provides for a method to refresh tokens over time, as needed.

In other words, the OIDC specification provides a native, web-friendly method for principal federation. It is a less secure method than client certificates, but generally secure enough for human principals to authenticate to websites and their corresponding APIs. OIDC is also a widely deployed framework for principal federation for workloads accessing server system APIs. For example, in order to authenticate a federated workload identity to Amazon Web Services [26].

1. The OIDC token can be subject to a forwarding attack, if the audience is coarse-grained. For example, if the audience is applicable to all authenticators in a trust domain, and principal P authenticates to authenticator A, then authenticator A can impersonate P for authenticator B in the same trust domain.

2. The OIDC token and other request content are not bound to each other. For example, if a request from principal P is destined for authenticator A, but the request transmits through proxy B, B might be able to change the request contents, or replay and mix-and-match other previously seen tokens.

3. The OIDC token practically cannot be revoked during its lifetime.

As part of the thesis work, I investigated what methods are available in order to mitigate some of the above methods. Myself and Micah Hausler presented a talk named *End to End Message Authenticity in Cloud Native Systems* on the topic at a conference in London [14]. The HTTP Message Signatures RFC [27] provides a method to mitigate forwarding attacks and bind request contents to the principal identity. The HTTP Message Signatures protocol can use either a symmetric or asymmetric key

for protecting request contents. For integration with the naturally distributed OIDC protocol, an asymmetric key is more suitable. There are two prominent methods for binding an asymmetric key to an OIDC token: the OAuth 2.0 Demonstrating Proof of Possession (DPoP) [28] and OpenPubKey [29].

2.1.3 SPIFFE

SPIFFE [30], an acronym for *Secure Production Identity Framework For Everyone*, is an identity and identity issuance standard focused on workload identities. SPIFFE aims define common semantics for principal federation across trust domains, and provide a standardized way for a workload to acquire an identity [31]. The concept of *trust domains* is built into the core of SPIFFE. A *SPIFFE ID* is a standardized identifier for a specific workload within a specific trust domain. At the time of writing, SPIFFE defines two credential types for carrying the SPIFFE ID: X.509 certificates and OIDC tokens [25].

SPIFFE was proposed by Joe Beda [32] in 2016, and it was inspired by Google's internal identity distribution framework, *the Low-Overhead Authentication Service* [33]. The SPIFFE project itself is purely a standard, however, there exists many implementations. One of the most popular open source implementations, is called SPIRE, which is an acronym for *SPIFFE Runtime Environment*. Both SPIFFE and SPIRE are graduated Cloud Native Computing Foundation (CNCF) [34] projects.

With regards to credential distribution, it is natural for a human to use a browser-based login flow to a human resources (HR) system in order to acquire a credential, but not so much for a workload. However, a workload always exists in some surrounding environment, and the environment "knows" the identity of its workloads. Thus is the idea of SPIFFE that the workload can rely on the environment always providing it a credential in a dynamic and standardized way [31].

Finally, SPIFFE provides a framework for managing the lifecycle of a federation relationship between an authenticator and an issuer [35], especially in the context of key rotation.

2.2 Authorization Engine

According to Jøsang [36], there is ambiguity regarding the definitions of and differences between the terms authorization and access control. In some contexts, the word authorization means "granting rights". For example, RFC 2196 [37] defines that "Authorization refers to the process of granting privileges to processes and, ultimately, users." and the general dictionary Merriam-Webster defines authorization as "to endorse, empower, justify, or permit by or as if by some recognized or proper authority" [38]. However, NIST SP 800-162 [39] makes no distinction between *authorization* and *access control*: "access control or authorization, on the other hand, is the decision to permit or deny a subject access to system objects". The NIST SP 800-162 definition suggests that authorization means "granting access".

To resolve this, Jøsang proposes that "Authorization is the specification of access policies" (*granting rights*) and "Access control is the enforcement of access policies"

(*granting access*). This is consistent with the ISO/IEC 27000:2018 [40] standard's definition of access control as a "means to ensure that access to assets is authorized and restricted based on business and security requirements". In other words, authorization (*granting rights*) is a prerequisite for access control (*granting access, based on granted rights*). The definitions of these words matter, both for consistent usage in this thesis, and as they are used to further define key information security concepts. Here are three examples from the ISO/IEC 27000:2018 standard [40]:

- *attack*: "attempt to destroy, expose, alter, disable, steal or gain unauthorized access to or make unauthorized use of an asset"
- *availability*: "property of being accessible and usable on demand by an authorized entity"
- *confidentiality*: "property that information is not made available or disclosed to unauthorized individuals, entities, or processes."

The most common authorization *request* can be formalized as: "Can principal P perform action A on resource R, in context C?" [41]. The authorization *decision* corresponding to a given request can be one of: *unconditionally allowed*, *unconditionally denied*, or *conditionally allowed*. A conditional allow is most often seen in situations where not enough data is available to produce an unconditional allow or deny.

The *authorization engine* or *authorizer* [10] is a system component responsible for *computing* an authorization decision from a request. On an abstract level, the authorization engine needs three types of input to compute the decision:

1. *The authorization request*: Information about the principal, action, resource and context.
2. *The authorization policies*: Grants of rights to principals, or *authorizations*. Written by a *policy author*, which often is a trust domain administrator. Usually the policy is role-, relation-, or attribute-based, see Section 2.4.
3. *The authorization data*: A policy might depend on a pieces of data, such as the age of a principal or a transaction amount. For data related to the principal, it might be convenient to include authorization data in the principal's credential, but this is generally considered a bad practice, as it scales poorly [43].

There are five core requirements for an authorization engine [9]:

1. *Expressiveness*: Policy authors need to model their access control after their organization and/or software system, which due to their variation from context to context, usually requires extensive expressiveness.

[10] Terminology used as synonyms and consistently in this thesis. However, in some sources, this is referred to as a *Policy Decision Point* (PDP) [42]

2. *Safety*: From the authorization engine's perspective, policies might be written by authorized but untrusted authors. This means that evaluating an untrusted policy must be safe for the engine. For example, policy evaluation should be side-effect free and use bounded resources.

3. *Performance*: Every request should be subject to an authorization check[11], which means that the time and resources that are required to compute the decision is added to every request. In addition, the engine might need to fetch policies and/or data, which should be taken into account.

4. *Analyzability*: Due to authorization being a safety-critical function, ensuring that the system functions as expected is paramount. Being able to introspect policies, and understand the semantic meaning of them, may prove critical.

5. *Correctness*: The engine implementation itself should indeed be correct. However, as implementing an authorization engine is not easy, it is easy for there being subtle bugs. Heavy testing of the engine implementation is pivotal.

Providing the policy author with good primitives for writing policies[12] is critical, as the top API security weakness, as determined by OWASP, is *Broken Object Level Authorization* [3].

2.3 Reference Monitor

The *reference monitor*[13][44] system component is responsible for *enforcing* decisions computed by the authorization engine. In other words, the reference monitor ensures that only authorized requests can be executed. The reference monitor should satisfy the requirements defined for the authorization engine, and in addition, be *non-bypassable* and *tamper-proof* [44].

The reference monitor making use of an authorization engine that fulfills its own requirements goes a long way in building a good reference monitor. That the reference monitor should be non-bypassable means that all system requests must be subject to the access control, and unauthorized requests are rejected. That the reference monitor should be tamper-proof means that it should withstand malicious tampering, such as privilege escalation attempts. An example of this is that even though some principal has the privilege to change privileges in the system, the principal must not be able to expand their own privileges that way.

It is a best practice to logically separate the authorization engine from the reference monitor. Not tightly coupling the enforcement code logic with the authorization decision computation allows for better separation of concerns, auditability, sharing, versioning and updates [9]. The OpenID AuthZEN Authorization API [45] is a proposal for a standardized interface between the reference monitor and authorization engine.

[11] Modulo it often making sense to cache decisions for a small amount of time, if decisions are expensive to compute or might have availability consequences.

[12] In the context of this thesis, the word *policy* implicitly refers to authorization policies.

[13] Terminology used consistently in this thesis; [42] refers to this as a *Policy Enforcement Point* (PEP)

2.4 Authorization Paradigms

2.4.1 Early Access Control Paradigms

As computers became accessible by multiple principals at a time (multi-tenant), there needed to be a way to manage access. The first model developed was the *access control matrix* [46], which exhaustively enumerates all allowed principal-action-resource tuples. However, that model does not scale well with an increasing number of principals and resources. *Discretionary Access Control* (DAC) [44] is an access control paradigm in which principals own resources in a decentralized manner. The owner can grant or revoke rights to its owned resources to other principals, at the owner's *discretion*. The creator of an object usually becomes the owner. Unix permissions and Google Drive are example implementations of DAC. Contrast this with the Mandatory Access Control (MAC) [44] paradigm, where there is no concept of an owner, other than the system as a whole. In MAC, a set of (usually few) administrators have centralized rights to change the authorizations in the system.

Harrison et al. [47] found in 1976 that if a protection system makes it possible for a principal to grant their rights recursively forwards, then it is not possible for the protection system to guarantee that a "bad authorization state" is not reached. This holds, regardless of the initial authorization state. This is due to there being an encoding of rights grant actions into a Turing machine, and asking whether a Turing machine will reach a certain state is undecidable and known as the *halting problem* [48].

Inspired by military applications, Multi-level Security (MLS) policies were also developed. MLS policies use a concept of *clearance levels*. The Bell-LaPadula MLS model [44] focuses on preserving confidentiality, by restricting information flow like follows:

- The *no-read-up* policy enforces that the principal clearance level must be higher or equal to any read object's level.
- The *no-write-down* policy enforces that the principal can only write to documents with a level higher or equal to any object being simultaneously read.

With this historical context covered, the next sections will focus on the paradigms used most commonly in contemporary server systems.

2.4.2 Role-Based Access Control

In 1992, Ferraiolo and Kuhn [49] introduced the concept of Role-Based Access Control (RBAC). Like MAC, the core assumption is that the system, or organization, is the owner of the data, not individual principals. A *role* authorizes principals bound to it to perform a set of enumerated `action-resource` requests. Principals getting access solely through roles eases on- and offboarding, increases uniformity in access, and makes updating the authorizations for everyone with a given role easy.

RBAC as proposed in [49] allows at most one active role at any given time, but I have not seen any real-world examples of that, except [26]. However, it is certainly a

good idea to not activate roles that allow performing destructive actions by default. For example, Github requires the principal to enter their password again, before destructive actions may be executed in a temporarily, higher-privileged "sudo mode".

However, RBAC has a complication called *role explosion* [44]. Role explosion appears when the authorization needs to be fine-grained. A common fine-grained policy requires multiple attributes with $\wedge$ semantics. For example, there might need to exist a role for `job=surgeon` $\wedge$ `city=helsinki` $\wedge$`specialization=jaw`, in order to follow the least-privilege principle. The worst-case number of roles required for n ANDed attributes $A_i = \{v_1, v_2, ..., v_m\}$ is $\prod_{i=1}^{n} |A_i|$.

Even if the number of roles would stay manageable, it is impractical to for a given role explicitly enumerate every authorized resource instance and corresponding actions. Often, the names of the resources are also unknown up front. Instead of explicit enumeration, today roles are usually attached to some predicate function, which define whether a resource instance "belongs to" the role or not. However, predicate functions over principal or resources are characteristic for Attribute-based Access Control (ABAC), which actually is a superset of RBAC. This due to *role* effectively only being a set-valued attribute of the principal.

2.4.3 Relation-Based Access Control

One evolution and superset of RBAC is Relation-based Access Control (ReBAC). ReBAC encodes the authorization decision problem into a graph reachability problem. Two data models for this have been proposed: Next Generation Access Control (NGAC) [42] and Google Zanzibar [50]. NGAC was not considered further for this thesis project work, as no production-ready implementation of it exists as open source. Google Zanzibar is Google's internal, consistent and global authorization system, which powers hundreds of services. Zanzibar scales to trillions of access control lists, and millons of authorization requests per second, while maintaining a 95th-percentile latency of less than 10 milliseconds. Multiple Zanzibar-inspired authorization engines have been implemented in open source. The two most prominent are SpiceDB [51] and OpenFGA [52]. Zanzibar supports a typed and flexible graph schema. The policy author can define the following:

Node and edge types:

- **Graph node types**: For example: principal, folder, and document. Graph nodes contain just two pieces of data: the type and an identifier.
- **Valid relations between node types**: For example: "principal and folder nodes are related through the 'read' relation", or "folder and document nodes are related through the 'parent' relation"

How to evaluate paths in the graph:

- **Same-type relation inheritance**: Allows defining that "any editor of a document is also a reader of the document".

- **Cross-type inheritance**: Allows defining that "any editor of a folder that is a parent of this document, is an editor of this document".
- **Set operations**: Allows composing all other rules with union, intersection and exclusion.

In the Zanzibar, there is only one data type, a *tuple*. A tuple represents a directed edge in the graph. The graph is typed, so a node is fully-qualified through specifying both the node type, and an identifier scoped to that node type. There can be many possible types of *relations* (represented as edges) between two nodes of the same type. Expressed in terms of our abstract model: the schema is the authorization *policy*, and the graph itself is the authorization *data*. Computing an authorization decision is done through a Zanzibar *check*, which starts a graph search from the *resource* node to the *principal* node. Check requests can in practice be fast, even though the graph is large [53]. This thanks to dividing the problem into smaller sub-problems that execute in parallel, and subproblem caching.

One can think of the Zanzibar graph as a highly optimized index of the concretely enumerated authorization state. Having this index makes it straightforward to lookup "what resources can principal P perform action A on?" and "what principals can perform action A on resource R?". Alternatively, a check request can return what paths in the graph were searched, which can aid debugging. The Zanzibar model fits many applications very well, and improves greatly upon RBAC. Especially any application that is oriented around sharing, but where the schema is fixed, is a great fit.

However, no authorization paradigm is a silver bullet, and Zanzibar is no exception. Due to the explicit enumeration from some intermediate node in the graph, to the final resource node, it is unnatural to express $\wedge$ and attribute-heavy policies, just like for RBAC. It is hard to check whether a resource node would be authorized, if it existed in the graph. At least without actually temporarily inserting the node, thus breaking caching, and turning a cheap read request into an expensive write. Also in general is there write amplification when every every potential write to the database might also have to change state in the Zanzibar graph.

Zanzibar is not a great fit if more expressiveness than $\in$ or $\notin$ is needed, e.g. string pattern matching, integer comparisons, or special data types like dates and durations. It might be possible to build an automated process, which inserts "special" nodes for custom data types needed. However, the more logic that is built into that automated process, the harder it is to understand why a certain principal has a given permission by looking at the graph. When performing a lookup of all resources a given principal can access, the predicate can be trivial, but the amount of resources enormous. If a logic-based method would be used instead, the predicate could be pushed down into a database index, instead of items being processed one-by-one. Finally, Zanzibar is not a great fit if the same resources are subject to dynamically-changing administrator-authored policies, like is common in cloud management systems.

2.4.4 Attribute-Based Access Control

Attribute-based Access Control (ABAC) is an access control paradigm whose policies make use of attributes of the principal, resource and context. On an abstract level, there are two techniques for implementing ABAC: logic-based and graph-based [44]. The main graph-based method is NGAC [42]. The NGAC model is enumerative, and thus shares some of the same benefits and drawbacks as Zanzibar, even if they differ. However, NGAC is not covered in this thesis due to lack of open source implementations.

Instead, we consider logic-based methods in this section. Unlike RBAC and ReBAC that explicitly enumerate the principals and resources, ABAC is very flexible as it does not require such an enumeration. This means an ABAC authorization engine might not need to store as much data as its R(e)BAC counterparts. This at the expense of builtin enumeration capabilities in the authorization engine. Enumeration of a logic formula, even in the simple case, corresponds to the *boolean satisfiability problem* [44], and is thus NP-complete or harder. However, lookups might be supported in another way. If the authorization engine supports evaluating a logic-based policy partially in the presence of partial data, the remaining policy expression is called a *residual*. The residual might be used for filtering the set of all objects into the set of authorized ones in a more efficient way. For example, by turning the residual into a Structured Query Language (SQL) expression.

The ABAC policy being logic-based often allows reasoning about the "intent" of the policy more directly than looking at an RBAC role or Zanzibar schema. More expressive operators than equality and set-based ones might be supported. However, expressiveness is in tension with the other goals of the authorization engine: performance, safety, analyzability, and correctness. In particular, the policy should not become so expressive that it is Turing-complete. In such a case, the policy cannot be known to execute quickly, in fact it is undecidable if it would ever terminate [48]. As a consequence, a Turing-complete program is not analyzable. Finally, a Turing-complete program might in the worst case be unsafe to execute, for example, it could invoke an untrusted function, or consume an unbounded amount of resources. Balancing these contradictory goals is an interesting engineering challenge.

Some ABAC implementations support forming attribute values into a hierarchy. Logic-based ABAC policies are a good fit in enterprise applications, which need to support administrators crafting custom policies.

3 Control Planes of Cloud Management Systems

In the last decade, the paradigm of *cloud native computing* has become prevalent across the technology industry. The cloud native mindset has reshaped the industry, with an ecosystem of over 200 related projects, from over 280,000 contributors contributing almost 20 million contributions [34]. *Control planes* are at the core of the cloud native mindset. A control plane ensures technological systems an end user organization relies on for their business activities, e.g. a website, database, HR management system, functions at all times like configured.

3.1 Control Planes

Control planes are needed, as naturally systems both in the physical and digital world, become more chaotic over time [54]. Also, Google has noted that "Failures are the norm in large scale systems." [55]. Control planes help the administrators of the organization to combat failures and inevitable chaos by constantly *reconciling* the desired state defined by the administrators and the actual observed state [56]. In engineering, this is a well-studied area: control theory. Control planes in cloud management systems apply control theory paradigms to the ensure the digital world stays functional.

The control plane can be contrasted with the *data plane* [57], which in turn actually performs the business function needed. An example from networking would be the control plane managing network forwarding rules, but the data plane (hardware and software routers) consuming the forwarding rules as input, and actually "does the work" and forwards the network packets to their destination.

In this thesis, we will focus on control planes that are built as uniform Representational State Transfer (REST) API servers [58], and exchange data in a JSON or JSON-compatible format, in a resource-oriented way. On a high level, the control plane API stores the following pieces of data:

1. the user-declared *desired state* of the system being controlled,
2. the system-observed *actual state* of the system, and
3. (optionally) authorization policies, which define who has access to what data.

In order for the API server to be backwards-compatible with old clients, the API should support multiple schema versions of the API objects it serves to clients. This is needed to, for example, be able to upgrade the API server without disruption to clients [56].

In addition to the API server, there is one or many *controllers* (also sometimes referred to as *reconciliation loops*, or *operators*) [59], that strive to drive the actual state towards user-declared desired state by taking appropriate action. This thesis will focus on how control planes that control cloud management system, and their access control, but the same paradigm applies to any target system as well.

3.2 Kubernetes

Kubernetes [2] is a production-grade container orchestrator, a control plane for running distributed systems in an environment-agnostic way by orchestrating workloads (usually running in containers) across a set of servers. Kubernetes was created by Google and open sourced in the summer of 2014 [60]. Its architecture is influenced by the knowledge accumulated by operating server systems at internet-scale with predecessor systems Borg [55] and Omega [61] at Google. In 2015, Kubernetes was donated to the non-profit Cloud Native Computing Foundation (CNCF) [34], a subfoundation of Linux Foundation. Thanks to CNCF owning the Kubernetes immaterial property, companies can under this vendor-neutral umbrella collaboratively contribute to the project.

Kubernetes has an extensible, declarative, scalable and resilient architecture. Kubernetes is written in the Go programming language [62]. Kubernetes itself consists of a set of microservices separated into control and data planes. At the heart of the system is a generic REST API server, which is responsible for uniform authentication, authorization, admission control, flow control, payload versioning, and more. The API server stores data in the etcd database [63]. All control plane requests must go through the API server, which is a forcing function to make all APIs (also system-focused, for better and worse) transparent to and usable by users.

The API server is extensible. An administrator can register their custom API, with a given OpenAPI schema [64], using a CustomResourceDefinition (CRD) [65]. After registration, the API server will serve this *custom resource* just like any other native Kubernetes API. This extension mechanism can be used to build custom controllers that take their input from custom resources, and control other target systems that Kubernetes does not control by default [56].

API objects in Kubernetes are *self-describing* through always encoding the information needed for anyone to decode the object, called `TypeMeta`. TypeMeta defines the API object:

- *kind* of object this is (type information)
- API *group* the kind/type is part of (to avoid naming conflicts between similarly-named kinds)
- API *version* of the type schema the object is serialized in

The Kubernetes API is a uniform rest API server, that supports the following logical HTTP request types and corresponding paths:

- For *list*, *watch*, *create*, *deletecollection*:
  `/apis/<group>/<version>/<resource>`
- For *get*, *update*, *patch*, *delete*:
  `/apis/<group>/<version>/<resource>/<object-name>`

The <resource> parameter in the URL denotes the type of the served API resource. In most cases, it is basically the same as the API object's *kind* in plural, for example, for Kubernetes `kind=Service` ⇒ `resource=services`. For a given group-version-resource tuple in an API server, there is exactly one group-version-kind.

Kubernetes `resources` also expose zero or many `subresources`, accessible at path `/apis/<group>/<version>/<resource>/<object-name>/<subresource>`. The most common subresource in Kubernetes is `status`, which allows an authorized principal to just update the actual state (`.status` portion) of the API object [65]. This is useful when giving controllers the ability to sync back the actual state of object, without giving the controller the ability to override the desired state (that a human wrote). There are also special *connectable subresources* that allow using the API server as a reverse proxy. Connecting to such a subresource allows opening a bidirectional stream of data to the entity being represented by the API object, for example, proxying network traffic from/to a workload in Kubernetes.

Group-version-resource to group-version-kind mappings may be queried from the API server's *discovery documents*, accessible e.g. at URLs, `/` and `/apis/<group>/<version>`. The group-version-kind denotes what API objects can be read from and written to the group-version-resource endpoint. Kubernetes exposes an OpenAPI schema for each group-version-kind. Thanks to these features, can Kubernetes clients be fully generic, with no custom per-resource logic.

Kubernetes has two types of scopes for resource types: *Cluster* or *Namespaced*. A *Kubernetes namespace* is a lightweight isolation mechanism for grouping API objects together within a single Kubernetes cluster. A namespace is meant to partition one Kubernetes cluster into a set of smaller trust domains, to which principals can be assigned access.

On top of every Kubernetes API object containing TypeMeta information, virtually every API object also contains `ObjectMeta`, which defines metadata for every object at JSON path `.metadata`. A cluster-scoped resource is fully-qualified within the cluster only by its *API object name*, always specified in the `.metadata.name` JSON property. A namespace-scoped resource is fully-qualified by the namespace (`.metadata.namespace`) and name, together. A namespaced resource adds a segment to the URL as follows: `/apis/<group>/<version>/namespaces/<namespace>/<resource>`. The name (or name+namespace combination) identifier is unique for a given point in time (across space), but not across time. In other words, two cluster-scoped API objects `foo` cannot exist at the same time, but the `foo` object can be deleted, and later another object can be created with the same name.

Most Kubernetes objects that are stored also contain the top-level properties `spec` and `status`, for the desired and actual state, respectively. Every object following uniform conventions allows for building generic features, such as understanding the condition of an API object, by inspecting the `.status.conditions` list [66].

List, watch and deletecollection requests may specify label and/or field *selectors* in the HTTP query parameters. ObjectMeta has a *labels* property, which is a string-string map. A label selector contains the following information: one label key, one operator, and zero or more label values (depending on the operator). Zero-value operators are

"exists" and "not exists", which match the label key being set to any value, or not being present. Single-value operators are =, ≠. Multi-value operators targeting any number of values are ∈, ∉. If multiple label selectors are specified in a request, they are logically ANDed. This is illustrated in Figure 1.

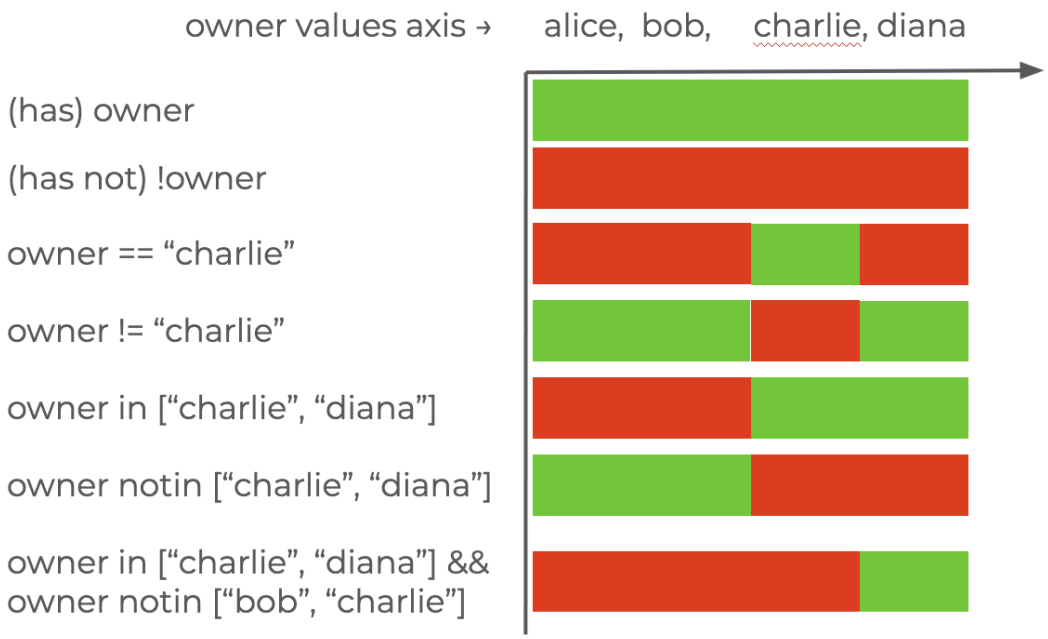


Figure 1: Visualization of selectors on the vertical axis, and matched label values in green (and not matched values in red) on the horizontal axis. Last example contains the ANDed result of two label selectors.

Field selectors are more limited, supporting only the =, ≠ operators, and a limited set of keys, referring to field paths of an API object, by convention using JSONPath [67] syntax. What field paths are supported, is dependent on the resource type and schema version. [68] contains a list of supported field selector paths in core Kubernetes types. As of recent versions of Kubernetes, CustomResourceDefinitions can define their own field selectors too. CRD field selector keys must be a simple JSONPath which refers to a string, integer or boolean field. If multiple field selectors are specified in a request, they are logically ANDed. If both label and field selectors are present, are their restrictions also logically ANDed.

With extensibility and uniformity being at the very core of the project, Kubernetes can practically be split into two layers: a "generic" part with control theory primitives, and a "workload" part, where Kubernetes practically implements workload orchestration APIs and controllers. In this thesis, we will focus on how to secure Kubernetes generic part, for the benefit of Kubernetes itself, and anyone building on top, like Crossplane.

Crossplane [7] is an example of a project which builds on top of Kubernetes' control plane framework, and adds capabilities for controlling cloud resources external to core

Kubernetes. The most popular cloud provider resources controlled with Crossplane through declarative Kubernetes APIs are Amazon Web Services, Microsoft Azure, or Google Cloud Platform resources. However, Crossplane is not tied to any specific cloud provider, is also extensible to support any cloud provider.

3.3 Securing a Control Plane API server

The traits of a modern API server implementing the most common security features are illustrated in Figure 2.

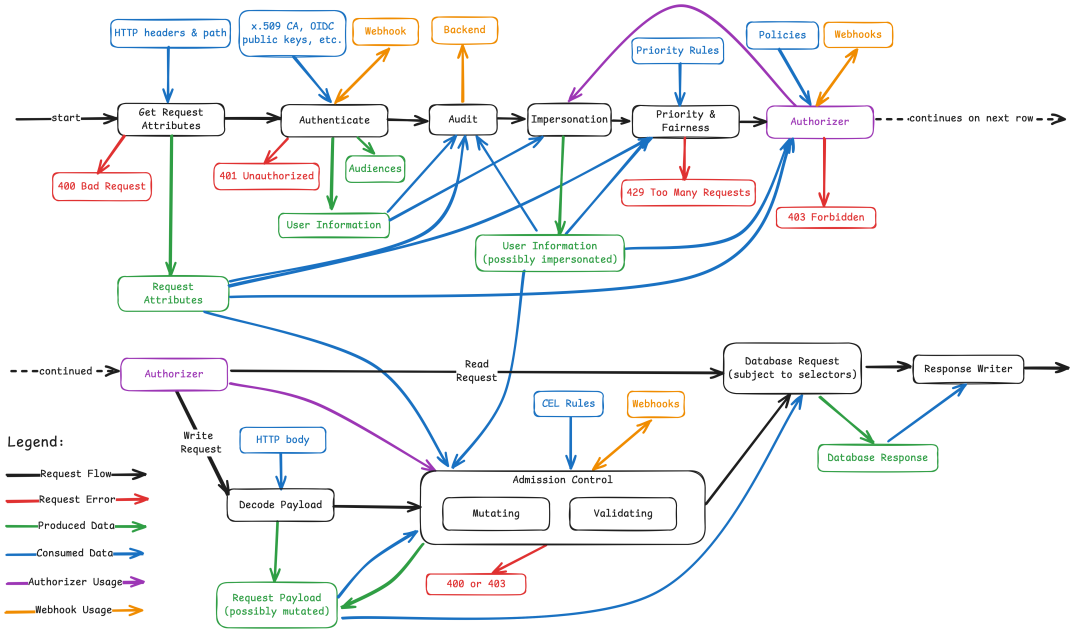


Figure 2: High-level flow for writing a secure API server, modelled after Kubernetes' generic API server

In order for the API server to function as a *reference monitor*, every request should be subject to the following steps, at least:

- *Authentication*: The request principal must be identified. If requests without credentials are possible, there should be a well-known principal for unauthenticated requests.
- *Auditing*: Knowing who did what when is crucial to be able to analyze afterwards what has happened in the system, especially if there are reasons to suspect unauthorized activity in the system.
- *Prioritization*: Not every request should be prioritized equally. Top priority should be requests from system components, such that the system itself can keep safely running, even amidst shortage of resources. Secondly, should not a single principal be able to use all API server CPU time and RAM, something

that easily could lead to intentional or unintentional Denial of Service attacks [69]. Thus should API server time and resources be distributed fairly, according to configured rules [70]. This mitigates the fourth most impactful OWASP API security flaw *Unrestricted Resource Consumption* [3].

- *Authorization*: The principal must have access to perform the given action, on the requested resource. At this stage, the request body is by design not decoded, in order to avoid Denial of Service attacks, where an attacker repeatedly sends huge payloads the API server would decode, before realizing that the principal is not authorized to perform the request.

If impersonation is supported, the API server must make sure that the original principal is authorized to impersonate the requested principal. The audit log should include both the original and impersonated principal.

Write requests should also be subject to validation, and in some cases even mutation, according to administrator policies (e.g. for compliance). In Kubernetes, the payload validation and mutation process is called *admission control*. The admission control stage has access to complete data; the principal, request attributes, payload, and the authorizer. This makes the admission control stage suitable for computing and enforcing advanced, fine-grained access control, if needed. Admission control can be used to mitigate the third most impactful OWASP API security flaw *Broken Object Property Level Authorization* [3].

On the abstract level, the API server is a reference monitor, as it enforces the authorization engine's decision. In addition, it is non-bypassable, as the API server is the only component which has access to the database etcd. It relies on the authorization engines to be tamper-proof (e.g. prevent privilege escalation), and be expressive, correct, safe, fast, and analyzable as needed.

The next section discusses Kubernetes specific implementation tradeoffs with regards to this generic model.

3.4 Kubernetes Authentication

Kubernetes' model of a principal contains the following information [71]:

- User name (string): Required. Usernames must be unique at any point in time (across space), but might be later re-used (not across time).
- Groups (string set): Optional.
- Unique ID (UID) (string): Optional. UIDs must be unique across both space and time.
- Extra (map from string to string set): Optional. Used by authenticators to propagate custom principal information to authorizers downstream.

Notably, Kubernetes does not provide any user management, but requires user management lifecycle to be brought by the environment. Kubernetes uses many different issuers of credentials for various purposes. A Kubernetes cluster forms its own trust domain. A user can introspect how they are authenticated using the SelfSubjectReview API, for example, for debugging. Kubernetes is usually configured with authenticators supporting the following credentials:

1. **Client certificates**: Kubernetes has a per-cluster Certificate Authority, which issues x.509 client certificates through the CertificateSigningRequest (CSR) API. Should only be used by Kubernetes system components only [14]. Not revocable.

2. **Kubernetes ServiceAccounts**: Kubernetes API server-issued, cluster-scoped, and OIDC-compliant JWT tokens. ServiceAccounts are automatically distributed to workloads running in the cluster. The Kubernetes API server publishes the OIDC discovery document, such that arbitrary authenticators can validate ServiceAccount tokens. Should only be used by workloads running in Kubernetes. All ServiceAccount tokens can be instantly revoked (for Kubernetes API usage) by deleting the ServiceAccount API object from the Kubernetes API.

3. **External OIDC tokens**: Kubernetes can be configured to trust and source user information from an OIDC provider [15], which manages the list of principals and their lifecycle. This practice is sometimes referred to as *Single Sign On*. The recommended way for humans to authenticate to the Kubernetes API server. Multi-cluster by design, as the principal is not scoped to a cluster.

However, if a feature not supported by the above authenticators is required, are there also two authentication extension mechanisms:

4. **Token Webhook**: A synchronous webhook, which resolves an unrecognized HTTP Authorization Bearer token to the UserInfo model using the TokenReview API. Responses may be cached for a specified duration.

5. **Authenticating Front Proxy**: Authentication can be performed by a front proxy in the request path before the request hits the API server. In this case, Kubernetes can source the user information from HTTP headers, given that the front proxy proves HTTP header authenticity to Kubernetes using mutual TLS.

[14] However, there is no partition of "system APIs" and "user APIs", so technically a normal Kubernetes could issue themselves a client certificate using the CSR API, if they had enough privileges (the cluster administrator usually does).

[15] Note that this is slightly different from how popular cloud providers (AWS, Google Cloud, Microsoft Azure) source federated principal information from external OIDC providers. Cloud providers usually force the client to exchange its externally-signed OIDC token for a cloud provider-signed credential [26]. Kubernetes just consumes the externally-signed OIDC token as-is, according to configuration [71].

Methods 3, 4, and 5 can all be used to make authentication uniform across Kubernetes cluster, paving the way for multi-cluster access control as well. Which method to choose depends on the specific use-case, and the consistency and availability requirements in particular. If immediate revocability support is not required, then method 3 is probably the easiest to implement. In hosted Kubernetes offerings, methods 4 and 5 are seldom available, and method 3 might be possible with limited configurability only. In such a case, an *impersonating* front proxy like [72] might be an option.

Interestingly, there is no native support for or usage of SPIFFE in Kubernetes, most likely due to Kubernetes implementing most of these features before SPIFFE was production ready. However, it would be an interesting future project to investigate whether there would be benefits to integrate SPIFFE and Kubernetes more deeply, as SPIFFE provides powerful multi-cluster trust federation capabilities.

One complication with Kubernetes authentication design is that all authenticators[16] share the same space of UserInfo names. This makes it technically possible for two logically distinct principals to both be authenticated with the same username and/or group, by two distinct authenticators. This configuration error would make it impossible for the authorizer to see the difference between the two users. This is a good example of how authentication and authorization must be designed together, and not in isolation. To avoid this error, the cluster administrator can for instance make sure that:

- no non-core authenticator can return any UserInfo string with the prefix "system:", that Kubernetes uses and reserves, and
- every authenticator uses a unique prefix for all returned UserInfo strings.

3.5 Kubernetes Access Control

As can be seen in Figure 2, in order for a write request to succeed, it is first required that the request is authorized. Then, the request payload is decoded and possibly mutated by mutating admission controllers. Finally, there is a set of validating admission controllers that can deny the request, if it is invalid, or not authorized. If there are no validating admission controllers, the request is implicitly allowed. Read requests do not go through admission control.

As with authenticators, Kubernetes supports multiple authorizers. The authorizers are organized as an ordered list, where each authorizer might produce an `Allow`, `Deny`, or `NoOpinion` response. The API server serially invokes each authorizer in order, and short-circuits whenever the response is either `Allow` or `Deny`. If response is `NoOpinion`, the next authorizer in queried, until there are no more authorizers. At least one authorizer must return `Allow` for the request to be allowed. Because order matters, it is critical to design the chain such that all `Deny` portions of all available authorizers are ordered before any authorizer that return `Allow`, if "deny trumps allow" semantics are desired.

Kubernetes provides an API called *SubjectAccessReview* (SAR) [73], which answers the most basic authorization question: "can principal P perform action A on

[16] Kubernetes uses multiple simultaneously; by default at least methods 1 and 2.

resource R?". There is also a *SelfSubjectAccessReview* (SelfSAR) variant, which is the same as a SubjectAccessReview, but for the authenticated principal. Any principal can perform a SelfSAR to discover their privileges, but principals should generally not have access to perform SAR requests for other users, as this might leak information. At the abstract level, SubjectAccessReview is the interface between the reference monitor (the API server) and the authorization engine. Kubernetes supports the following authorizer modes:

1. RBAC: Authorizes principals' access based on roles and bindings registered declaratively in the Kubernetes API.

2. Node: A system-only, ReBAC-based authorizer for giving Kubernetes node agents (kubelets) access to exactly the API objects referencing the given node, transitively. This limits the blast radius of a compromised node agent.

3. Webhook: Like the authentication webhook, Kubernetes can send a synchronous SubjectAccessReview to some number of webhook implementations. The response can be cached for a configurable amount of time.

Usually, hosted Kubernetes offerings do not allow configuring the authorizer chain, which means that the only available mode is RBAC. There are multiple reasons, but they are primarily safety- and performance-related. A slow webhook authorizer would slow down the whole cluster. A broken webhook could render the Kubernetes cluster unavailable or completely insecure.

One particularly interesting side-effect of authorizing `watch` requests [2], is that they are long-lived (by design), but authorization is only enforced when opening the watch, not throughout the lifetime. This means that a principal can get access to new information, even after they were revoked access, as long as they managed to open a watch before revocation.

Common use-cases for validating admission control include the following:

1. Semantic API validation, defined by the API author. For example, when the validation requires inputs from multiple fields. Does usually not reference the principal.

2. Compliance restrictions, defined by the policy author (administrator). Enforces that the shape of all API objects of a certain type follow certain standards. For example, never allow root containers in the cluster. Does not usually reference the principal.

3. Compound Authorization [74], defined by the API and/or policy author. For example, issue a SubjectAccessReview for a "virtual permission" (as discussed in Section 3.5.1), depending on a protected field on the payload object. Another use-case is authorizing references. For example, a well-known privilege escalation is that even though a principal might not be able to read a Kubernetes Secret from the API, the principal can read the Secret through creating a workload that references the Secret. References the principal in the SubjectAccessReview.

4. Conditional Authorization, defined by the policy author. Similar to compound authorization, but hard-codes some of the principal information, in order to restrict what payloads a given principal is able to submit. For example, only allow administrators to set a specific `adminAccess` property [75].

Kubernetes has no builtin notion of an "owner" of an API object, and is thus not a DAC system. On the contrary, is Kubernetes RBAC structured as a classical MAC system, where only a few policy authors set the rules for all objects. Even though some principal can write to an API object, they most likely cannot change the permissions related to it. A Kubernetes namespace forms a trust domain. Kubernetes Special Interest Group Auth (SIG Auth) does not recommend[17] splitting namespaces into smaller parts, as that easily becomes unmanageable. Instead, it is recommended to create another namespace for the other principal, if possible. In a similar manner, it is recommended to design the API objects such that they map to exactly one type of persona. This leads, in the best case, to the simple case where fine-grained conditional authorization is not needed.

However, it is not always possible to avoid the need for conditional authorization. In that case, the fact that the policy author must first "over-grant" in authorization, and then "take away" the permission in admission, is not very user-friendly. See Figure 3 for an example. Being able to specify this use-case with just a single, uniform policy object would be preferable, abstracting away the technical authorization-admission split. However, this thesis does not propose any changes to uses of admission control for API validation and/or compliance use-cases.

There are a set of core admission controllers built into the Kubernetes API server. In addition, admission controllers can be added by the user in the following ways:

- ValidatingAdmissionPolicy: An API that empowers API or policy authors to define their admission validation rules using Common Expression Language (CEL). The benefit of this method include that the API server does not have to perform any webhook, thus not operationally depending on another service's availability.

- Webhook: In case more expressiveness is needed, the API server can send a webhook to some HTTP server registered with the API server. However, if the webhook is unavailable, this may cause an availabiliy concern for the API server.

There are four admission actions: `create`, `update`, `delete`, and `connect`. The *patch* action in authorization becomes either an admission `create` or `update`. The *deletecollection* action in authorization becomes a single-item admission `delete`, for every matched API object. Although logical, understanding the relation between these different stages' actions adds to the training needed for the policy author.

[17] I am not sure whether this is documented anywhere, other than possibly in the public and recorded video meetings, and/or how the APIs in fact have been designed so far. The source for this information is practically through discussing with the SIG Auth leads at KubeCons.

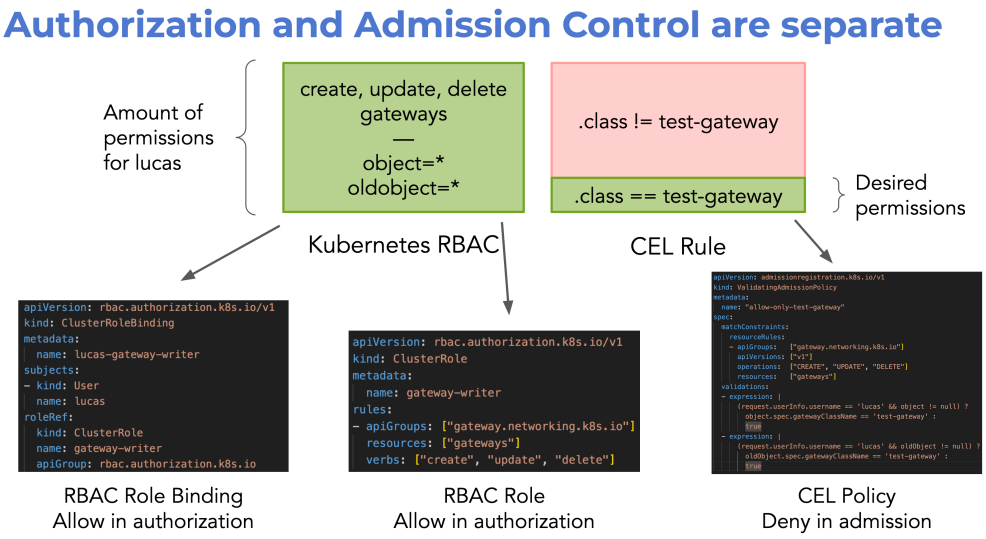


Figure 3: An example of conditional authorization. The principal 'lucas' should only be able to create Gateway objects when field `.spec.gatewayClassName == 'test-gateway'`. In order to authorize this, three API objects are needed. Image source: [76]

3.5.1 Kubernetes RBAC

The primary way to configure authorization in Kubernetes is through Kubernetes RBAC [77]. The RBAC feature allows the administrator to create *roles*, and through *bindings* assign a role to a set of *subjects*. All RBAC configuration is stored in the API server. A subject, or in this thesis' terminology *principal*, can be one of:

- kind=User, which targets the *username* field of the authenticated user's UserInfo.
- kind=ServiceAccount, which is a shorthand for username `system:serviceaccount:<serviceaccount-namespace>:<serviceaccount-name>`
- kind=Group, which targets the *groups* set of the authenticated user's UserInfo

An RBAC limitation is that the UserInfo UID and extra information cannot be bound to from a role binding. If the implementation was "classic RBAC", as discussed in Section 2.4.2, the role definition would enumerate exactly every API object and actions allowed for that API object. However, this would be extremely tedious and error prone, so a Kubernetes role actually matches objects to give bound principals access to in an *attribute-based* fashion. A role contains multiple ORed *rules*, each of which selects API objects according to the following five dimensions: API group, resource (type), subresource, namespace, and name. For each of the dimensions, the policy author can express "match any value for this dimension". In order for an object to be matched,

it need to match every dimension in a given rule. Furthermore, every rule defines what authorization actions the bound principals should be authorized to perform. A visualization of how Kubernetes RBAC is structured is shown in Figure 4.

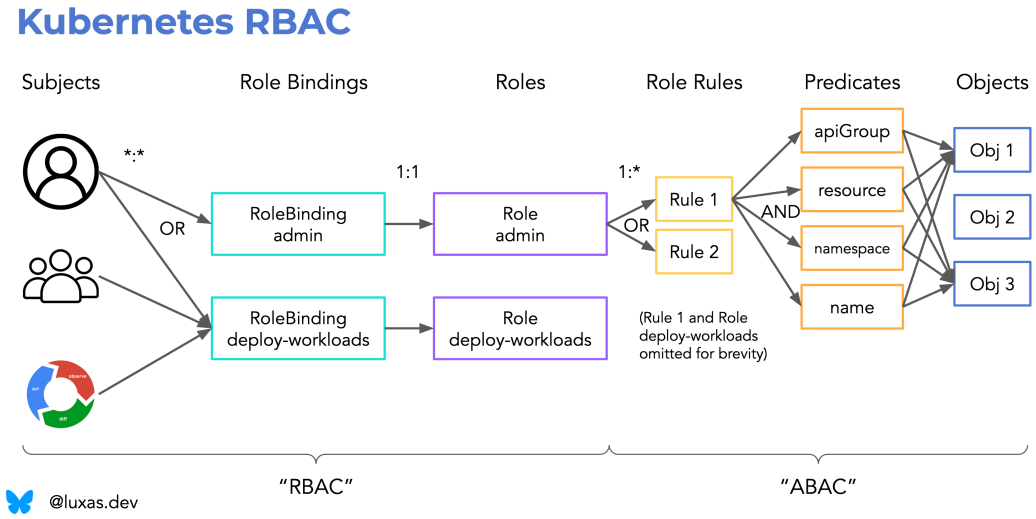


Figure 4: Kubernetes RBAC visualized. Image source: [76]

RBAC is an extensible API with no schema nor validation. A core Kubernetes idea is that creation order of API objects should not matter. In this case it means that applying a role referring to a CRD API, which has not yet been created, must not be blocked. Also, Kubernetes authorization is not limited to groups, resources, and actions that actually are served by the API. It is possible to register "virtual permissions" for any API group, resource, subresource and action, even though they do not exist in the server. This pattern is actually used for compound authorization, which we will discuss in Section 3.5.2. However, the lack of feedback can make RBAC hard to debug.

Notably, RBAC does not allow the policy author to specify labels and/or field selectors. An example case where this easily becomes a practical problem is for management of Kubernetes *Secrets*. A Kubernetes Secret is a Kubernetes API, which allows storing secret data in the API server. The Secret can be encrypted when stored in etcd, but whenever it is retrieved from the API server by someone with enough RBAC permissions, the secret data is sent to the principal in plaintext. Thus, are Kubernetes Secrets not considered very "secure" [78] [6], as they are just one RBAC mistake away from exposure. RBAC only supports allow rules, by design, but deny rules and/or finer-grained authorization could help solve this issue.

One way finer-grained access control is implemented in Kubernetes today, is through special subresources [79]. However, that is not a recommended pattern anymore [79]. The pattern, on a high-level is such that the API server registers a per-resource custom Hypertext Transfer Protocol (HTTP) handler, and restricts just

changing certain properties of the API object targeted. This makes it practically possible to give a principal access to only change certain fields, by only giving them access to the subresource. However, custom subresources are not supported for CRDs, which means that this pattern is not available to platforms building on top of Kubernetes.

Kubernetes RBAC supports privilege escalation prevention, as recommended. In other words, even though a principal is authorized to `create rbac.authorization.io rolebindings`, they cannot bind themselves (or anyone else) to a role which has more permissions than the principal currently has. This is enforced by the API server.

3.5.2 Kubernetes ValidatingAdmissionPolicy with CEL

Common Expression Language (CEL) [8] is an open source expression language from Google, used heavily in various parts of Kubernetes. CEL is not focused on authorization specifically, but instead aims to be a generic and extensible expression language. CEL has some good properties which makes it a good fit for executing as part of an API server request. Recall the criteria defined for an authorization engine in Section 2.2:

- Expressiveness: CEL is fairly expressive. It is built upon the [80] data model. It has JSON-compatible types, loops, enums, and more.
- Safety: CEL is side-effect free, and memory-safe.
- Performance: Loops are guaranteed to terminate, and bounds can be put on for how long expressions can evaluate.
- Analyzability: CEL can be analyzed to some extent, through parsing the AST.

Kubernetes uses CEL in the ValidatingAdmissionPolicy [81] API, which allows policy authors to write validating admission control logic, without running a webhook server. This is good from an availability point of view. From an auditing point of view, reading a CEL expression is more transparent than seeing a webhook to some opaque server. The ValidatingAdmissionPolicy API allows for compound authorization, as the authorizer is accessible as a function in the CEL environment.

4 Computing Authorization Decisions with Formal Logic

This chapter covers how authorization decisions can be computed, and the computation process itself can be verified, using formal logic.

4.1 Formal Reasoning and Verification

In security-critical applications, safety becomes a top priority. Safety might mean that

- no bad program state is reached,
- a result is eventually produced, that is, the program does not deadlock, and/or
- any error occurring makes the system *fail safely*

In any real-world system of interest, the number of states is far larger than the human brain can comprehend, and thus often design for. Writing ad-hoc end-to-end, integration or unit tests for common use-cases is of course *necessary* for the programmer to verify that their implementation matches the specification, however, tests alone are often not *sufficient*, as they are usually not exhaustive.

Another key issue here is that in order for a program to be implemented correctly according to a well-understood specification, it is required that the specification is unambiguous. This is usually not the case without extensive engineering effort, as business requirements are usually worded informally and at a high level.

However, a *formal specification* describes the system in a way that a machine can understand and use for automated and systemic analysis and testing, even for complex properties like deadlocks and failing safely. Lean [82] is such a specification language, that even supports compiling the specification into executable code for use in differential testing. Differential testing is the method in which the programmer:

- writes a specification of how the program should work (detailing the semantics, without any optimizations) in a specification language like Lean,
- writes an optimized implementation of the program (which is then actually used),
- uses the specification to systemically generate millions of test cases, then finally,
- runs a test case both against the model specification and the implementation, and verifies they match. If they do not, a bug has been found.

Determining whether a principal can perform a given action on a resource, is a classic safety-critical problem, where the programmer really wants to make sure their specification and implementation is complete. If the system reaches a bad state, it might give someone unauthorized access to resources in error. If the system can deterministically be tricked into a deadlock, it can become a target of a denial of service

attack. If the system does not fail safely, protected resources could, for example, be exposed if a system dependency cannot be reached. The following sections cover how an authorization problem can be modelled in a formal way.

4.2 Satisfiability Modulo Theories

As mentioned in Section 2.4.4, Turing-complete programs are generally not very suitable for writing authorization policies, as the programs cannot be known to halt, be side-effect-free, or be analyzed. On the contrary, one of the popular, but least expressive logic forms is *propositional logic* [83]. In propositional logic there is only *propositions* (or variables) that either take value true or false, and *relations* between the propositions that take the form of constraints, through formulas. There are two common normal forms, called conjunction normal form (CNF) and disjunction normal form (DNF). CNF and DNF are ways to organize the formula, consisting only of the elementary binary boolean operators AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). Other binary operators like implies ($\rightarrow$), equivalence ($\leftrightarrow$) and similar can be reduced to the elementary ones in a straightforward way.

The Boolean satisfiability problem (afterwards referred to as "SAT") of whether there is a *model*, that is, a mapping from the variables to a boolean such that all constraints are satisfied (evaluate to true), is the canonical NP-complete problem [84]. SAT being NP-complete means that both that the problem itself belongs to the NP complexity class and that any other problem in the NP complexity class can be reduced in polynomial time to SAT by a deterministic Turing machine. SAT belonging to the NP complexity class means that a *non-deterministic* Turing machine could solve the SAT problem in polynomial time, and a *deterministic* Turing machine can verify a solution in polynomial time. At worst, the runtime of SAT is exponential in the number of variables ($O(2^n)$) in the worst case, as there being 2^n states the boolean variables could be in. A CNF formula consists of *literals* (which is a boolean variable or its negation) and a set of clauses, which are a disjunction of literals (e.g. $l_1 \vee l_2 ... \vee l_k$). The formula is in CNF form then if it is a conjunction of clauses (e.g. $C_1 \wedge C_2 \wedge ... \wedge C_m$ [83].

Expressing authorization problems in propositional logic has some good properties. For instance, the authorization problem is safe for a service provider to execute, analyzable, and fast (given the number of variables is reasonable).

However, there are many practical use-cases where propositional logic is not expressive enough. Policy authors usually want some way to express strings, integers, sets and the relevant operators of those primitives. All of which are non-ergonomic to express using only boolean variables for a user. Satisfiability Modulo Theories (SMT) extend SAT with concepts from first-order logic such as $\forall$ and $\exists$ semantics and a type system (usually called many-sorted first-order logic) [83]. A given *theory* can interpret expressions of non-Boolean data types such as strings and integers into, eventually, Boolean values. One complication of including the first-order logic quantifiers $\forall$ and $\exists$, is that they make the SMT satisfiability problem undecidable [85]; which violates the desired property of our authorization system that it shall be fast and analyzable. This puts practical limits of the expressiveness of the authorization language [86].

There are two common approaches to solve SMT expressions. The *eager* approach effectively converts expressions with non-Boolean data types statically into many Boolean variables. However, this does not practically work so well for real-world expressions, results in unnecessarily many new Boolean variables (hurting scalability), and does not allow the solver to exploit the higher-level semantics. The *lazy* approach instead uses a SAT solver augmented with a dedicated "theory solver", which on-demand converts expressions containing supported non-Boolean data types to a boolean value, or a conflict. Further, might the theory solver return conflict constraints or implications between multiple expressions (e.g. $(a = 1) \rightarrow \neg(a = 2)$ which turns into clause $(\neg x_{a=1} \wedge \neg x_{a=2})$, informing the solver that a cannot be both 1 and 2 at the same time).

SMT-LIB [87] is an initiative for SMT solver developers to standardize on the input/output interfaces of solvers, develop new and better theories, collect standardized benchmark collections, and connect users, developers and researchers in the area. Popular SMT solvers include Z3 [88] and cvc4/cvc5 [89].

4.3 Encoding Authorization Policies into SMT with Cedar

Cedar Policy, an Apache 2.0-licensed open source project from Amazon Web Services Inc., implements all three authorization paradigms discussed so far (ABAC, RBAC, ReBAC). Cedar aims to balance all four goals of the authorization system: expressiveness, performance, safety, and analyzability [9]. Cedar is carefully designed to be encodable into SMT. Cedar is effectively a stateless authorization engine library, written in Rust. Cedar can easily be integrated as a library in almost any other programming language through the Webassembly interface, thanks to Rust's good support for Webassembly [18]. The Cedar library computes an authorization decision (allow or deny) based on the following pieces of information given to it:

- *Request Attributes*: Principal, Action, Resource, and optional Context.
- *Entity Data*: Information about the "entities" in the system, that is, possible principals, resources, resource containers, and other ancillary entities (e.g. "teams"), and the relations between them. This data is usually queried from a database, normalized, and fed to Cedar.
- *Set of Policies*: Cedar supports two policy statement types: **permit** and **forbid**. A policy always has a *scope* which tells what principal(s), action(s), and resource(s) it applies to. Optionally, a policy might be further constrained with to **when** and **unless** conditions (where **unless** is just the negation of **when**)
- *Schema*: Cedar supports (optional, but highly recommended) static typing for the request, entities, and policies, in order to catch errors at "compile" time, and not runtime.

[18] More information about the integration can be found at https://www.rust-lang.org/what/wasm

4.3.1 The Cedar Policy Language

Principals and resources are "entities", object instantiations of a given type defined in the schema, with a unique identifier [90]. Entities might also have attributes attached to them. Supported data types in the Cedar schema include: Boolean, String, Long (a signed 64-bit integer), Entity, Record (think: object with string keys and values of any other arbitrary type), and Set (of some other primitive data type). In addition, are there four extension data types: *ipaddr*, *decimal*, *datetime*, and *duration*. Note that decimal is not a floating-point number, as the data type has fixed precision of four digits after the decimal. Floating point numbers are not supported as they are considered unfit for an authorization system due to their impreciseness [91].

```
// Policy 0: Any User can create a list
// and see what lists they own.
permit(
    principal,
    action in [Action::"CreateList",
               Action::"GetOwnedLists"],
    resource == Application::"TinyTodo");

// Policy 2: Admins can perform any action.
permit(
    principal in Team::"admin",
    action,
    resource in Application::"TinyTodo");

// Policy 4: Interns can't create task lists.
forbid(
    principal in Team::"interns",
    action == Action::"CreateList",
    resource == Application::"TinyTodo");
```

```
// Policy 1: Any User can perform any action
// on a List they own.
permit(principal, action, resource)
when {
    resource is List &&
    resource.owner == principal
};

// Policy 3: A User can see a List
// if they are either a reader or editor.
permit(
    principal,
    action == Action::"GetList",
    resource)
when {
    principal in resource.readers ||
    principal in resource.editors
};
```

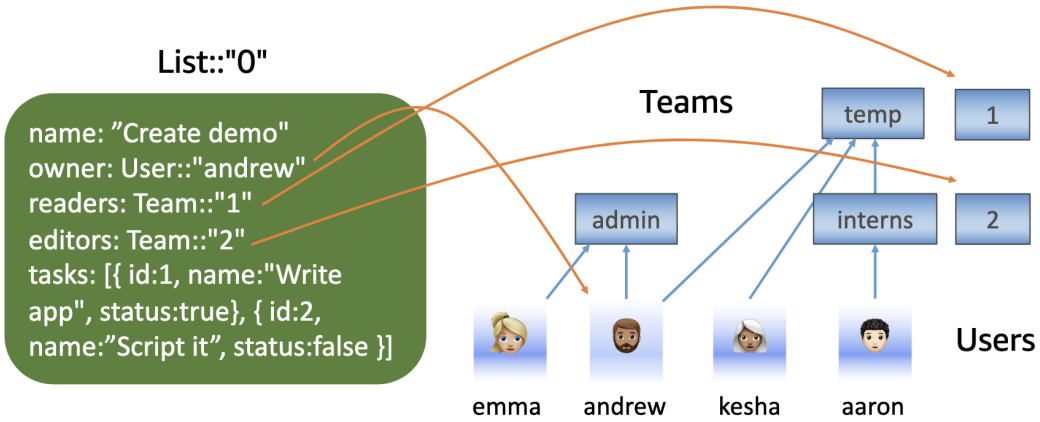


Figure 5: Examples of Cedar policies and a corresponding entity hierarchy, empowering policy authors to use ABAC, RBAC, or ReBAC. Image source: [9]

Thanks to entities having attributes, policy authors can write ABAC-style policies, such as "forbid employees that have worked less than two years at the company to read any resource whose confidentiality level is 'secret'", as illustrated in Figure 6. There

```
forbid (
    principal,  // applies to any principal
    action,     // applies to any action
    resource    // applies to any resource
) when {
    principal.yearsAtCompany < 2 &&
    resource.clearanceLevel == "secret"
};
```

Figure 6: Example of ABAC policy.

are two features that empowers ReBAC, where the first one is that relations between objects can be modelled through an entity attribute, whose value is another entity reference. The "owner" attribute of the List in Policy 1 in Figure 5 is an example of that. The second empowering feature is showcased in Policy 3 in Figure 5; entities can actually be related to each other in a hierarchical fashion, in the form of a directed acyclic graph (DAG). The schema defines what types can be parents/ancestors of a given type. As discussed earlier, RBAC being a subset of ReBAC makes Cedar support RBAC with ease, for example showcased in Policy 2 in Figure 5. Cedar performs transitive closure of the DAG before policy evaluation for the convenience of the policy author, and to be able to make 'principal in Team::"temp"' lookups in $O(1)$ time. Cedar best practices recommend that each policy follows at most one of the ABAC, RBAC or ReBAC patterns, as combining them "are at best confusing and at worst error prone" [92].

One interesting best practice that follows when entities are ordered into a graph, is that every resource should belong to some resource container entity [93], which is distinct from the "owning" user identity entity [94], as a very common use-case is to allow delegating access across user/team/organization "owner" boundaries, eventually, as illustrated in Figure 7. Further, note that policies can be "attached" to any point in the resource hierarchy, e.g. not necessarily always at the "top" level, instead at whatever level of the hierarchy that makes sense. In the Figure 7 case, any of the "Account", "Folder" or "File" levels can thus serve as "policy attachment points".

Actions may also be grouped into a hierarchy, e.g. such that there are both fine-grained actions such as "create", "update", and "delete", as well as coarser-grained ones, like "write", which could include all those three actions, enabling policy authors to in a user-friendly way choose their level of granularity. Authorization requests to Cedar would always use the most fine-grained applicable action.

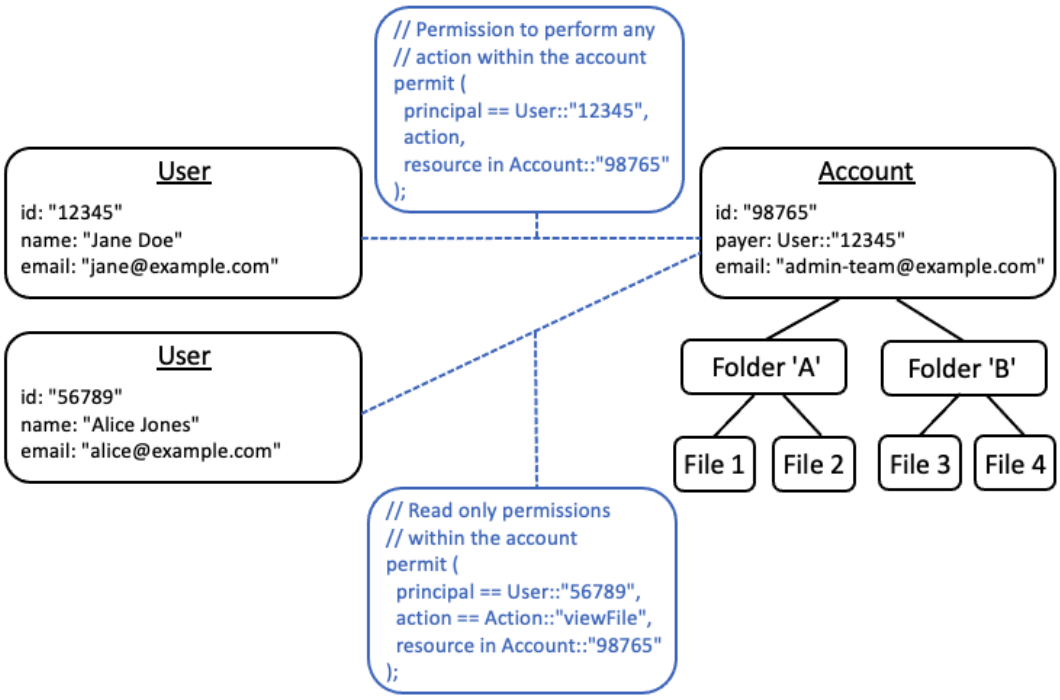


Figure 7: Best practice showing that a) resource containers should be used, b) the user identity shall be a separate entity than the resource container. Image source: [94]

4.3.2 The Cedar Policy Authorizer, Evaluator and Validator

One interesting detail about Cedar is that while its policies can be mapped to SMT expressions [9] [95], actually no SMT solver is used in Cedar, but the policy evaluator is built "by hand" in Cedar's Rust code, optimized for this use-case. At the time of writing, there is yet no open source Cedar-to-SMT compiler either [96], although AWS seems to have one internally [9]. One especially interesting detail in that paper is how the SMT encoding of the acyclity and transitivity of the DAG cannot use the $\forall$ quantifier, as that would make the encoding undecidable. Instead, the SMT encoder grounds the DAG constraints specifically for the finite set of entities referenced by policy expressions, so that decidability is maintained.

In order to verify the correctness of the hand-crafted evaluator for this purpose, differential testing is used [82], as discussed in the previous section, to verify the optimized Cedar evaluator implementation fulfills the specification. The key properties are:

- *Deny by default*: If no permit policy evaluates to true, the request is denied.
- *Explicit allow*: If a request is allowed, then at least one permit policy evaluated to true.
- *Forbid trumps permit*: If any forbid policy evaluates to true, the request is denied.

The Cedar schema defines all possible actions exhaustively, and exactly what principal and resource types are applicable in the context of what action. This is important for two reasons:

- *Policy Validation*: The policy validator ensures that the policy is *well-formed* for all possible request permutations of principal, action, and principal types (called "request environments"). The validation should be *sound*, that is, ensure that if a set of policies is validated successfully for a given schema, there are no runtime errors originating from the schema.
- *Policy Slicing*: The Cedar authorizer does not have to consider every policy given to it, only the *sliced* subset of policies that may apply to the current request environment. This optimization leads to higher performance as the number of policies grow, thus better scalability. However, Policy Slicing has not been implemented in core Cedar yet, as there has not been a clear need for it so far, however, AWS has an internal implementation of it.

Note that in theory, any entity type can be both a principal and resource (even at the same time!), there is no "special" or hard-coded partition between principals and resources. The only restrictions that are put are those that the schema author defines on what types might be principals and resources in the context of a given action. However, both recently mentioned features must be verified (e.g. through differential testing) to be *sound*:

- *Policy Validation* is sound if it is guaranteed that if a set of policies is validated successfully for a given schema, there are no runtime errors originating from the schema.
- *Policy Slicing* is sound if the returned authorization decision is always the same with slicing as without; that is, the no excluded policy applied to the request.

Normal usage of Cedar usually makes the following things happen internally:

1. The user tells Cedar to parse the Cedar schema in DSL or JSON format into the internal schema representation.
2. The user tells Cedar to parse the Cedar policies in Cedar DSL or JSON External Syntax Tree (EST) format into an internal Abstract Syntax Tree (AST) representation, that can readily be converted into an internal *Expression* type.
3. Optionally, the user may choose to validate the policy set against the schema before proceeding.
4. The user tells Cedar to parse the entity data (and whether to validate it according to the schema) into concrete *Value* types.
5. The user tells Cedar to authorize a given request against the given entities and policy set. Internally the *Authorizer* builds an *Evaluator* from the request, entities and policy set.

6. The authorizer uses the evaluator to evaluate the expression of every policy in the policy set. The result can be true, false, an error, or a *residual expression* [97], if there was not enough data to fully determine a value.

7. The authorizer skips all policies that errored [91], regardless of if they were permit or forbid policies, and either returns a concrete response (allow or deny) along with diagnostics on what policies contributed to the decision or errored, or a partial response with the residual expressions of the policies that could not fully be evaluated with the data given.

How can runtime errors happen although the policy validation was sound and valid for the policy set, entity data and request? One simple example of a runtime error that is not easily spotted during static analysis time is integer overflow errors caused by arithmetic operations. The reason Cedar does not fail the request completely (by default), is that such behavior could lead to Denial of Service attacks [91]. If the client application wants, however, it can look at the diagnostics information in the Cedar response, and choose to fail closed in any case.

5 Uniform Kubernetes Access Control with Cedar

This Master's thesis proof of concept implementation of a Cedar + Kubernetes integration builds upon and extends Micah Hausler's *Cedar Access Control for Kubernetes* proof of concept project [10]. The improvements I propose here, are meant to be open sourced and merged into the upstream [10] project, once that repository has moved to the `https://github.com/cedar-policy` organization. The Cedar project as a whole is also in the process of being donated to the Cloud Native Computing Foundation [98], to be able to collaborate on the project in a vendor-neutral way and grow the community, without one single company owning all the immaterial property. Thus transitively, is the aim of these contributions to be donated to the CNCF. The implementation of my ideas are in this thesis referred to as "the Cedar integration" or "thesis project".

5.1 Project Goals and Requirements

Most of the requirements have already been mentioned throughout the background material. However, a summarized list of high-level, long-term goals for the project, grouped by user persona, as follows:
For resource owners/policy authors:

1. **Uniformity across access control stages**: Making authorization and admission control uniform [99]. Write one "right-sized" policy from the get-go, instead of multiple policies in two different languages (e.g. Kubernetes RBAC and CEL). Provide a similar experience for both reads and writes.

2. **Familiarity and Continuity**: Use the same concepts as Kubernetes RBAC and CEL-like syntax, so users do not have to change mindset. Support converting e.g. existing RBAC rules into this encoding to get a smooth transition.

3. **Linting**: Thanks to the typed schema, improve the policy author experience by catching errors (typos, common misunderstandings, logical errors) *before* the policy being applied to the cluster. Expose an experience for common editors, e.g. Visual Studio Code.

4. **Principal querying**: Support answering the query "What resources can this principal access?". One variant of this query should be for principal referring to the currently authenticated principal, allowing for self-introspection and debugging.

5. **Resource querying**: Support answering the query "What principals can access this resource?". Both of these querying capabilities are also useful for auditing/debugging purposes.

6. **Atomicity**: Simplify access control orchestration by either an access control policy being in force, or not. This in comparison to today, where one logical policy might be partially applied.

7. **Deny policies**: Deny policies should be used with utmost care, but they have their place. Introducing deny policies are a rather complex task, but one should investigate the viability. Clever use of *policy tiering* might make deny policies easier to use.

8. **Policy comparisons**: Empower policy authors to compare policies to each other, to find a partial order between policies [100], [101], to e.g. evaluate policy equality, find no-operation policies, and prevent privilege escalation.

For hosting providers:

9. **Safety and Performance**: The reason hosted Kubernetes offerings (e.g. GKE, AKS, EKS) do not support webhook authorizers today, is primarily safety- and performance-related. A broken webhook authorizer could render the Kubernetes cluster broken or completely insecure.

For API authors:

10. **Better parity with native types**: Today, native builtin Kubernetes API type authors have an advantage over CustomResourceDefinition authors, in that builtin types might encode custom access control logic, for example, in the storage layer. For generic control plane use-cases like Crossplane, it would be good level the playing field for between CRDs and native types. More expressive, unified and conditional authorization can be of help here.

For the project:

11. **Parity**: Keep most or all functionality from current solutions (up to analyzability constraints). For example, implement privilege escalation prevention and the SubjectAccessReview API.

12. **Multi-cluster encoding**: Ideally, a policy author would be able to write a policy, that can apply to multiple clusters at once. This is a good example of a touchpoint between authentication and authorization; in order for this to work, authentication semantics must be uniform across all clusters. Also, all entity IDs must then either be globally unique UIDs, or be scoped under the cluster ID.

13. **Use a general purpose engine**: Implementing an access control engine that satisfies all these requirements is a lot of work. I do not think it is sustainable for the Kubernetes community to re-implement its own custom, specialized, but still feature-rich access control engine. Instead, it is preferable to work with some general purpose engine, such that features Kubernetes needs are implemented in the engine, benefiting other users, and vice versa.

For Kubernetes upstream:

14. **Investigate Conditional Authorization**: Write a Kubernetes Enhancement Proposal [102], that would investigate what it would take to make a Kubernetes SubjectAccessReview [73] response conditional.

15. **Investigate Unified Semantics for Label- and Field Selectors**: Today, field selectors are just an opaque string, that does not necessarily map to the JSONPath of the property selected by the object (although it usually by convention does) [68]. In addition, field selectors do not support "in" semantics, like label selectors do [103]. Finally, label and field selector predicates are always specified separately, and ANDed together, instead of allowing to write one unified selector expression incorporating both pieces of data in an arbitrary way. This could also be investigated in a Kubernetes Enhancement Proposal as an extension of this work.

These goals are aspirational, and vastly exceed the scope of one Master's thesis (project). This thesis focuses on exploring the viability and direction of the solution space, and implementing a proof of concept demonstrating the viability of the initial direction. It should be noted that this list is a superset of the original goals of [10].

5.1.1 Authorization Engine Requirements

In addition to the requirements defined in Section 2.2, what additional requirements does the above specification put on the underlying authorization engine used? If Kubernetes is to get the implement the defined features, either it needs to use something off-the-shelf, or implement its own engine conforming to the following specifications:

1. **Typed schema**: To give users a good experience, users should be able to write and validate their policies against a typed schema. This is also a prerequisite for most other requirements defined here.

2. **ABAC support**: The engine must be able to evaluate expressions with at least the following JSON data types that exist in Kubernetes: strings, booleans, integers, objects, and sets of any of those primitive types. On top of this, the following logic operators should be supported, preferably in any order: $\wedge$, $\vee$, $\neg$, $\in$, $=$, and whether a (typed) object *has* a given property, to avoid the so-called billion dollar mistake [104].

3. **SMT compilation**: There should exist a process for translating a policy into a logically equivalent SMT expression, which can be used to check for e.g. logical consequence or equivalence.

4. **Deny policies**: Deny policies need to be supported in the language, deny should be the default, and a deny policy should trump an allow policy.

5. **(Typed) Partial Evaluation**: In order to know in the Kubernetes authorization stage, when the request payload is not yet decoded, whether the request *can* become authorized, the engine must be able to work with incomplete data, and respond with either of: unconditional allow or deny (no matter the request payload), or conditional allow (along with predicates that need to evaluate to true for the request payload). Partial Evaluation is also used for principal and resource querying. [97]

Optional, but nice features of the authorization engine used for the control plane API access control use-case:

1. **Language Server Protocol [105] Support**: This in order to support a nice e.g. Visual Studio Code experience, validating policies as one writes them.

2. **ReBAC support**: Relation-based Access Control is useful for some paradigms, especially the closer one gets to "normal application data". However, the focus of this thesis is cloud infrastructure management, where ABAC-style policies are more predominant.

3. **Ability to infer data needed for policy set**: Even though the schema of possible data a policy might authorize against is vast, usually just a small fraction of that data actually influence a given authorization request. This means, it would be inefficient to evaluate a request against thousands or millions of entities, if just 10-20 are needed. *Entity slicing* [106], [107] is a solution to this, and can also aid policy linting, e.g. knowing that a policy accesses some piece of data, but not another data piece.

The Cedar policy language and specification [9], and associated Rust library implementation satisfies most of the above set requirements for the authorization engine.

During the thesis research process, I did not find other open source authorization engines that were fit for the task. For example, I also evaluated the Zanzibar [50] data model, and its open source implementations SpiceDB [51] and OpenFGA [52]. I managed to reduce Kubernetes RBAC into a Zanzibar graph. However, that did not yield the desired properties listed as requirements here. The main problem was lack of ABAC-style expressiveness. This led to most of the authorization logic moving from the graph, to the reconciliation from high-level policy author intent (e.g. an RBAC policy) and the edges. As the reconciliation loop is Turing-complete, it was hard to analyze policy author intent by looking at the graph. In addition, it was hard to support the SubjectAccessReview API without breaking caching by writing a "dummy" node representing the queried resource.

CEL was not chosen directly, as it does not (to my knowledge) have an encoding into analyzable SMT logic, most likely as it supports loops. However, it is most likely possible to use CEL as a Cedar "frontend" for an expression, by restricting CEL's syntax to the intersection between CEL and Cedar expressiveness, and converting the CEL expression into Cedar. It is likely that something like this will be done, if Cedar is used with Kubernetes, as Kubernetes already integrates heavily with CEL.

At the time of writing, some of the required features in Cedar are still being actively worked on, e.g. Typed Partial Evaluation [97], policy compilation into SMT-LIB format [96], and entity slicing using data tries [106]. However, when given these features, I'm optimistic the community around Kubernetes and Cedar can implement a good experience for Kubernetes policy authoring using Cedar, satisfying most of the requirements defined in this chapter.

5.2 Integration Architecture

The integration is designed to be usable without changing Kubernetes core. On a high-level, the project is structured in the same way as [10], as the solution space is not very large. The architecture is visualized in Figure 8.

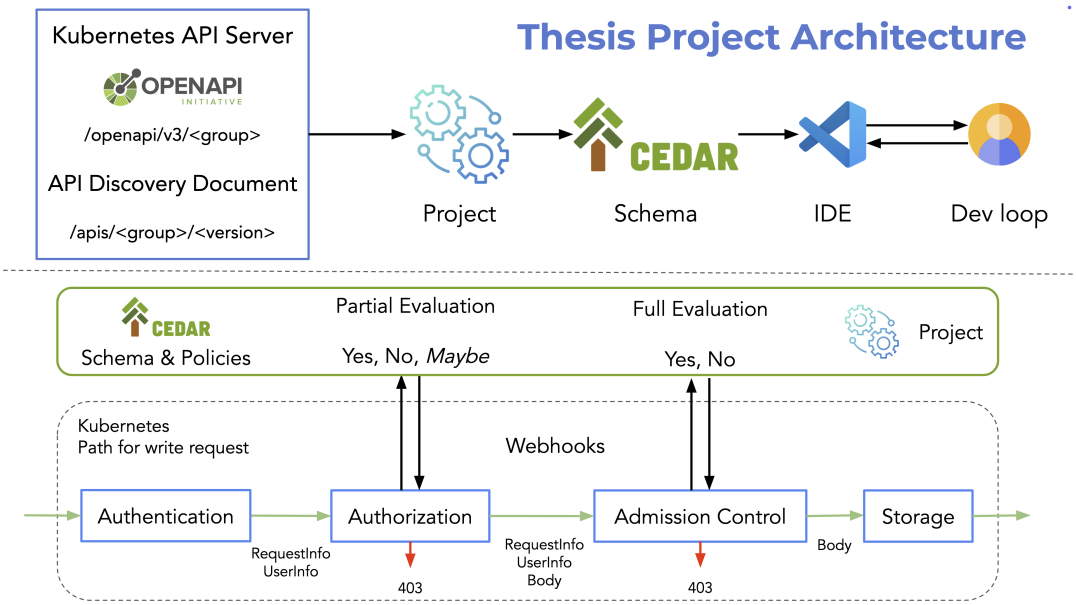


Figure 8: Project architecture. A typed Schema is generated from Kubernetes discovery and OpenAPI [64] data. Kubernetes webhooks to the project are registered for both the authorization and admission stage.

5.3 Schema Generation

The thesis project generates the typed Cedar schema dynamically from the Kubernetes' API discovery mechanism and the OpenAPI v3 [64] schema. An example is given in Appendix A. The discovery document describes what group-version-resources are registered with the API server, and what group-version-kind schema is supported for a given group-version-resource. The OpenAPI v3 document describes the API schema of a group-version-kind. This process works for both native Kubernetes APIs, and CustomResourceDefinitions.

For some APIs, there is a destinction between a field being unset versus set to the empty value (e.g. `0`, `""`, or `[]`). In order to distinguish such cases, OpenAPI uses the `required` list of non-nullable fields, which maps nicely to Cedar's `required` property of an object property.

One Cedar entity type is generated per group-version-resource. Common properties for all `resource` types are:

- `apiGroup`: Required, e.g. "storage.k8s.io",

- `apiVersion`: Required, e.g. "v1",
- `resourceCombined`: Required, e.g. "pods" or "pods/exec",
- `namespace`: Required if namespaced, non-existent otherwise
- `name`: Optional on top-level resources, required on subresources,
- `request`: Optional. Contains the payload of the request, for `create`, `update`, `patch`, and `connect` actions.
- `stored`: Optional. Contains the currently stored API object, for `list`, `watch`, `update`, `patch`, `delete`, `deletecollection`. Only a partial object available for `list` and `watch`, as discussed in Section 5.6.

Using typed entities allows for unifying authorization and admission constructs in a single `resource` object, unlike [10]. However, there is also one untyped, generic entity resource type called `k8s::Resource`, just like [10]. This means that the policy author can write untyped policies that only target authorization attributes, or typed policies, where the API object is also available. Technically, it means that the integration must allow the request if either a typed policy allowed it, or a typed one.

One key restriction of Cedar of note is the absence of loops [86] and "open records" or "embedded attribute maps" [108], depending on terminology. Loops and open records would improve expressivity, but would make analysis very hard or impossible, and thus were Cedar RFCs 21 [86], 66, and 68 [108] rejected.

Kubernetes makes use of open records primarily in the following places:

1. ObjectMeta `labels` (selectable) and `annotations` (non-selectable),
2. the UserInfo `extra` map propagated from the authenticator,
3. resource requests (where the untyped key can be e.g. `cpu`, `memory`, `nvidia.com/gpu`, etc.)
4. propagating unstructured parameters to e.g. scheduling or storage

Of these use-cases, representing the first two are the most critical, given the authorization use-case of this project, but also the latter two data points could be interesting from an authorization perspective. On top of this, Kubernetes semantically has three types of arrays [109]:

1. maps: the JSON array contains JSON objects, where one or more scalar fields of the object logically build a unique map key, through which the map can be indexed. Kubernetes API conventions [66] strongly recommends this pattern over open records represented directly in JSON. An array item might be updated without modifying the other array items.
2. sets: JSON array of unique scalar values

3. atomic lists: no restriction of item uniqueness, the list is always updated as a unit, and treated as a unit.

The schema generation process in [10] turns JSON arrays of item type `T` into Cedar `Set<T>`, and open records of value type `T` into a `Set<{key: String, val: T}>`. However, due to Cedar not having loops, Cedar sets being unordered, and the only set operations available are =, `contains`, `containsAny`, `containsAll`, there is not much utility for a policy author to do anything useful with Cedar sets when `T` is a non-scalar type.

This thesis proposes improvements to status quo by using a recently introduced Cedar feature, entity tags [110]. An open record with value type `T` is turned into Cedar `entity StringToTMap = { keys: Set<String>} tags T`. Cedar tag keys are always strings, that can safely be indexed through the `.hasTag(str-value)` and `.getTag(str-value)` methods available on the `StringToTMap` type. Further, can Kubernetes map-type JSON arrays also be indexed like a map in Cedar, using the same technique, empowering the policy author to write their policy against data contained in maps, e.g. container properties of a Kubernetes Pod. Finally, the `keys` set of the generated Cedar map type allows the policy author to force the map to be of bounded, well-known size, with well-known map keys only.

The thesis proposes to expose the HTTP verb-specific options exposed as query parameters to the Cedar action context. This allows policy authors to understand, for example, whether a write is a dry-run or not. Connect verbs in Kubernetes use query parameters to as the payload, as usually the subresource can be connected to using the HTTP GET verb, which usually does not carry a request body. Connect query parameters are mapped into `resource.request` in Cedar.

5.4 Policy Autocompletion and Validation

First and foremost, even if the policy author did not use any of the new features proposed, they might still get a nice experience writing their (existing) Kubernetes RBAC rules in Cedar. The first goal of this project is to re-implement Kubernetes RBAC in Cedar, which is already a decent amount of work, as Kubernetes RBAC is in itself no simple system. However, one of the added benefits of using Cedar for existing policies, is autocompletion and policy validation in Visual Studio Code [111].

The schema gives the policy author the opportunity to explore the whole API surface in one place. The Visual Studio Code integration also gives the option of jumping from a type reference to the type definition, a very useful feature, given that the Cedar schema generated just from Kubernetes core APIs yields a file around 5000 lines long. During the thesis project implementation, I noticed that the formatting of the Cedar schema file was sub-optimal, and thus contributed nicer formatting [12], to benefit readability.

Figure 9 shows two examples. In the first example the UI shows a dropdown of available property names according to the typed schema. The second example shows how the typed schema catches errors through static analysis already as the policy

authors write the policy. This in contrast to an untyped model like Kubernetes RBAC, which gives no feedback.

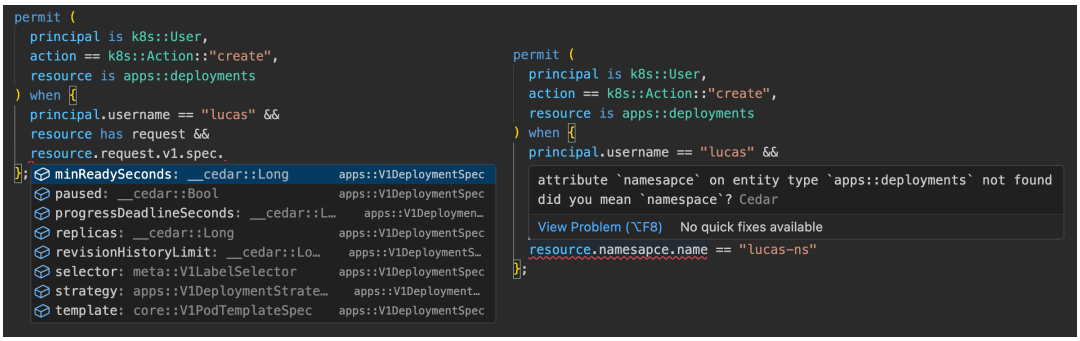


Figure 9: Example of how the Cedar language server helps the policy author. On the left, a dropdown shows what properties are available, when writing the policy. On the right, a user-friendly error is shown if the policy author made a mistake, such as misspelling a property name. The image has been slightly edited to fit the thesis format.

5.5 Partitioning by label and field selectors for writes

A long-standing Kubernetes user request has been to support label and/or field selectors in Kubernetes RBAC [6]. One common concrete use-case is to limit Gateway controllers to be able to work across the whole cluster, but only read and write Secrets with a given label. Gateway controllers proxy network traffic into the Kubernetes cluster from the outside world and are usually configured through Gateway API CRDs. They source the TLS certificates to use for e.g. terminating HTTPS from Kubernetes Secrets, and work across the whole cluster. However, according to defense-in-depth practices, they should not be able to access unrelated Secrets, as that could be an attack vector.

The feature has not been implemented due to many reasons, including:

1. The recommendation from Kubernetes Special Interest Group Auth (SIG Auth) in Kubernetes is to generally use namespaces and design the API object types with given personas in mind, as discussed in Section 3.5. Selector authorization is easy to misconfigure.

2. Inconsistencies in label and field selector availability across verbs. Today, list, watch and deletecollection verbs support selectors, others do not.

3. It could mean that the result of a given SubjectAccessReview (SAR) is not just dependent on the backing policy, but also the state of the data objects in the database. This could make caching SAR responses unsound, and thus a performance, correctness and availability concern. For example, authorization could (non-optimally) become dependent on data objects:

   (a) An object O has a label "owner=lucas", and the applicable authorization policy states that "owners can read their objects".
   (b) If the principal lucas is able to read O *without restricting the get, list or watch request with the "owner=lucas" label selector*, authorization has to check the contents of the database, to determine if O has the correct label, before sending it to lucas.

4. System-enforced immutability of selected labels and fields is preferred. Otherwise questions such as the following are raised:
   (a) "should someone be able to edit an object such that they cannot read it anymore?"
   (b) "should someone be able to edit an object they cannot read, in order to give them read access?"

5. RBAC upgradability concerns: If Kubernetes introduced RBAC labelSelector support in a new server version, and a new client sends an RBAC Role with labelSelectors specified to an old server, the labelSelector fields could be silently dropped, and the security posture would be weak, as if the labelSelector restriction was not applied.

6. All in all, it is a complex feature, both to use and to implement. It is highly unlikely Kubernetes could invent something sustainable by themselves in this area, as the features needed approach the generic authorization engine space.

The above issues clearly set boundaries for the solution space:

1. One may conclude that in order to make the experience that the users want to work, both the authorization and admission control stages of the API server need to work together.

2. The user must "know" how to limit their request up-front, before making the request, in order for authorization to not be dependent on the state in the database. In the example above, the use-case would be possible, if the principal lucas knew that they cannot read any object, but only objects where the label "owner=lucas" hold; and thus the read request must include this label selector, such that the authorization stage can based on this selector information determine whether the read request is allowed or not.

3. If this feature is implemented, the user experience should be top of mind, making it easy to do the right thing, and hard to misconfigure. In addition should it support acceptable version skew. It would probably be good with an easy-to-use, Kubernetes-native, best-practice label and field selector authorization API frontend, as well as a lower-level authorization language backend into which the frontend reduces.

4. In order for the system to understand whether a principal e.g. would not initially have access to read an object, but have access to edit the object such that the principal can afterwards read it, one need to be able to analyze the authorization language deeply.

5. Ideally, the API for authorizing a principal to read an object with a given label selector, should be as close as possible, if not identical, to authorizing a principal to write an object with the same label. This means that the policy author should not have to think about what is the difference between Kubernetes authorization and admission control stages.

This thesis explores a possible solution to this problem through the use of Cedar. Figures 10 and 11 give examples on how to use label- and field selectors more safely. The approach proposed by this thesis relies heavily on typed partial evaluation [97], which is still being implemented at the time of writing in Cedar. Currently in Cedar there exists an untyped Partial Evaluation feature, which shows the idea, but has some limitations making it unsuitable for production use. As part of the thesis, I fixed one bug in the untyped partial evaluation feature [11], to help this proof of concept. For the rest of the thesis though, *partial evaluation* refers to the typed variant. Let us explore what using the feature could look like.

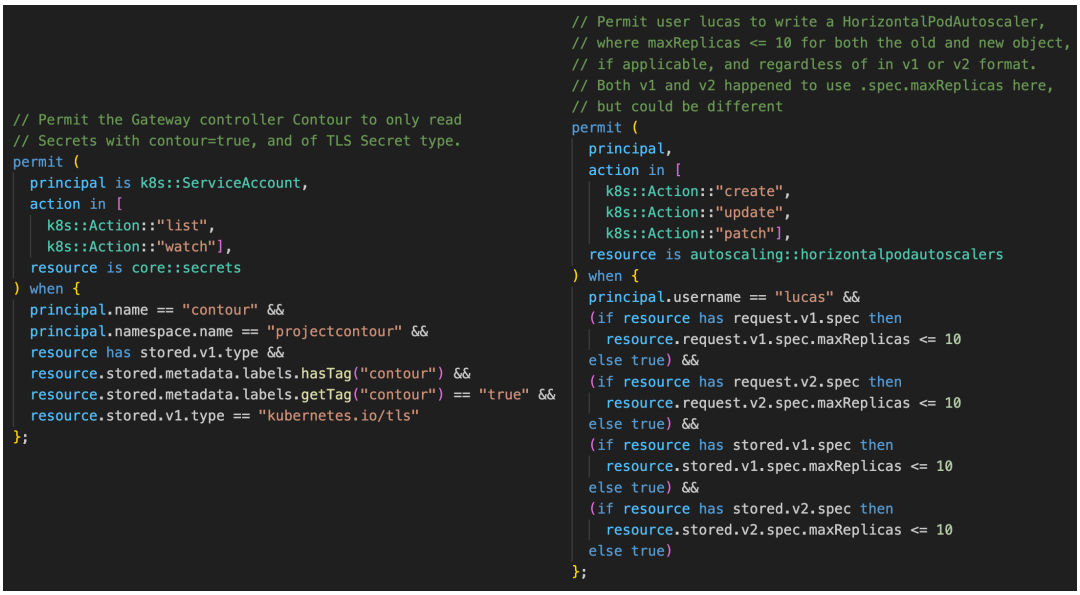
```
// Permit Kubernetes users in the "with-owner-labels" group to
// read such Secrets which they "own", across any namespace.
permit (
  principal is k8s::User,
  action in
    [k8s::Action::"list",
     k8s::Action::"watch",
     k8s::Action::"delete"],
  resource is core::secrets
) when {
  principal.groups.contains("with-owner-labels") &&
  resource has stored &&
  resource.stored.metadata.labels.hasTag("owner") &&
  resource.stored.metadata.labels.getTag("owner") == principal.username
};

// Permit Kubernetes users in the with-owner-labels group to only
// create Secrets which have the owner=<username> label
permit (
  principal,
  action in [k8s::Action::"create"],
  resource
) when {
  principal.groups.contains("with-owner-labels") &&
  resource has request.metadata &&
  resource.request.metadata.labels.hasTag("owner") &&
  resource.request.metadata.labels.getTag("owner") == principal.username
};
```

```
// Permit Kubernetes users in the with-owner-labels group to only
// update Secrets when both the old and new object has the owner label
permit (
  principal is k8s::User,
  action in
    [k8s::Action::"update",
     k8s::Action::"patch"],
  resource is core::secrets
) when {
  principal.groups.contains("with-owner-labels") &&
  resource has stored &&
  resource.stored.metadata.labels.hasTag("owner") &&
  resource.stored.metadata.labels.getTag("owner") == principal.username &&
  resource has request &&
  resource.request.metadata.labels.hasTag("owner") &&
  resource.request.metadata.labels.getTag("owner") == principal.username
};
```

Figure 10: The policy on the left gives certain users the ability to operate on Kubernetes Secrets, where there is an owner=<username> label, uniformly for reads and writes. The top-left policy says that an existing stored object must have the label, the bottom-left that create must set the label, and the right policy that an update must have and preserve the label. The same three policies could be written more concisely as well, as shown in Figure A1

As mentioned in Section 3.3, writes requests in Kubernetes are first authorized, and then subject to validating admission control. However, reads in Kubernetes are subject to only authorization, not admission. This makes it hard to integrate the user experience



```
// Permit the Gateway controller Contour to only read
// Secrets with contour=true, and of TLS Secret type.
permit (
  principal is k8s::ServiceAccount,
  action in [
    k8s::Action::"list",
    k8s::Action::"watch"],
  resource is core::secrets
) when {
  principal.name == "contour" &&
  principal.namespace.name == "projectcontour" &&
  resource has stored.v1.type &&
  resource.stored.metadata.labels.hasTag("contour") &&
  resource.stored.metadata.labels.getTag("contour") == "true" &&
  resource.stored.v1.type == "kubernetes.io/tls"
};
```

```
// Permit user lucas to write a HorizontalPodAutoscaler,
// where maxReplicas <= 10 for both the old and new object,
// if applicable, and regardless of in v1 or v2 format.
// Both v1 and v2 happened to use .spec.maxReplicas here,
// but could be different
permit (
  principal,
  action in [
    k8s::Action::"create",
    k8s::Action::"update",
    k8s::Action::"patch"],
  resource is autoscaling::horizontalpodautoscalers
) when {
  principal.username == "lucas" &&
  (if resource has request.v1.spec then
    resource.request.v1.spec.maxReplicas <= 10
  else true) &&
  (if resource has request.v2.spec then
    resource.request.v2.spec.maxReplicas <= 10
  else true) &&
  (if resource has stored.v1.spec then
    resource.stored.v1.spec.maxReplicas <= 10
  else true) &&
  (if resource has stored.v2.spec then
    resource.stored.v2.spec.maxReplicas <= 10
  else true)
};
```

Figure 11: The policy on the left shows an example of how a Gateway controller, Contour, could be configured to only be allowed to send read requests with labelSelector contour=true and fieldSelector type=TLS. The policy on the right shows how the policy can be written against multiple versions of the versioned object schema, as discussed in Section 3.1

for both types. This integration integrates with Kubernetes in both authorization and admission stages, as follows. The integration receives a SubjectAccessReview from Kubernetes, basically asking whether the principal is allowed to perform the action on the given resource. Notably, the resource itself is not sent with this request, only metadata. Policy example Figures 10 and 11 all target both the metadata of the request (apiGroup, resource type, namespace, name), *and* properties of the API object itself, e.g. labels or fields. In addition, Figure A1 demonstrates an example of a uniform label restriction policy, which applies to both reads and writes. At the authorization stage, however, the API object payload (if any) and the existing, stored API object in the database are both unknown.

This is where partial evaluation comes into play. Even though the request and stored objects are unknown, policies might be written such that the principal always or never would have access, that is, such that the result is an unconditional allow or deny. This is actually to be expected for the largest chunk of requests. However, we might also get a *conditional allow*. For example, if a permit policy is such that all expressions referring to the request metadata evaluated to *true*, and there is an un-evaluatable *residual* policy expression left referring to only the unknown API object(s), the integration should allow the request at this stage, and enforce the residual to hold true later, when the API objects are known. Figure 12 illustrates this.

Today, there is no method for a webhook authorizer to communicate to Kubernetes that the request is conditionally allowed. However, this is something that one could

```
// Allow team-a to only create storage of the slow-hdd class
permit (
  principal is k8s::User,
  action == k8s::Action::"create",
  resource is core::persistentvolumes
) when {
  principal.groups.contains("team-a") &&
  resource has request &&
  resource.request.v1.spec.storageClassName == "slow-hdd"
};
```

```
// Allow team-a to only create storage of the slow-hdd class
permit (
  true,   // principal type == k8s::User
  true,   // action == k8s::Action::"create"
  true    // resource type == core::persistentvolumes
) when {
  true && // ["team-a", "eng"].contains("team-a")
  true && // resource has request
  resource.request.v1.spec.storageClassName == "slow-hdd"
};
```

Figure 12: Example of a partially authorized request (on the right) of a policy (on the left) in the authorization stage, where the user in group "team-a" performs a create of a Persistent Volume. A create request always contains a request payload, but at the authorization stage, the request payload is unknown. Therefore, the last AND expression term cannot be evaluated. This illustrates a *conditional allow*, where the authorization response should be Allow, and the condition be enforced instead during admission control

explore in upstream Kubernetes, after this thesis project is finished. Until then, secondly, the Cedar integration registers itself as a webhook admission controller, and enforces an unconditional allow or deny there. At the admission stage, both the request and stored API objects are available in the AdmissionReview payload sent to the admission controller. Therefore, with complete data, all policies conformant with the schema can be evaluated fully.

This feature allows the policy author to right-size the policy they write, e.g. one where a principal can create an API object, only when a certain field has a certain value. For writes, this was possible before with Kubernetes RBAC (allowing creates for the principal) and ValidatingAdmissionPolicies (denying creates for the principal when the field value is not correct), but this was not a uniform nor atomic experience. See Figure 3 for an example.

5.6 Partitioning by label and field selectors for reads

What we have covered this far makes it possible to write policies that both target metadata and API object data in a uniform way, for write requests. How can we integrate a uniform experience also for reads? Admission control is not available for read requests. To begin with, the integration would perform a normal Cedar evaluation in the authorization stage, without attached API object data. If either of unconditional `Allow` or `Deny` responses are returned, this response is returned directly. If the evaluation returns a conditional residual expression, it means that some read requests within this scope are allowed, but not all. In this section, we will cover a method to compute a mechanism for turning the residual expression into a concrete `Allow` or `Deny` response.

As described in Section 3.2, *list*, *watch* and *deletecollection* requests may specify label and/or field selectors in the HTTP query parameters. There is an upcoming feature in Kubernetes to include the the label and field selectors in the attributes present in the authorization stage (that is, the SubjectAccessReview API) [112]. All

label and field selectors are ANDed. If there are duplicated selector requirements for the same key in the request, the requirements can trivially be folded into exactly one selector per key.

Kubernetes admission control allows the policy author to implement the generic `isValid(principal, newobject, oldobject)` interface, where `newobject` is the JSON object sent in the write request, and `oldobject` is the old object from storage. The Cedar mapping for write requests allow the policy author to implement the same object-scoped interface, with only names changed: `canWrite(principal, resource.request, resource.stored)`. This means that for reads, the policy author would expect to be able to implement a similar object-scoped interface `canRead(principal, resource.stored)`. To some extent, this is possible. However, only field selectable fields [68] can be provided in the object mapping in a read; a request referring to an unsupported field selector returns HTTP status code 400. The Cedar integration is responsible for ensuring the safety of the allowing a request implemented through this interface. Let there be n unique selector keys. As a given label key might not be defined for a given API object, the data model for a label value is either `None` or `Some(String)`. Let U be the infinite universe set of all possible values, including both `None` and all `Some(x)` strings. The integration determines that the request can be allowed when the safety criteria in Equation 1 holds.

$$
\begin{aligned}
&o = (v_1, v_2, ..., v_n) \in O_{all} = \prod_{i}^{n} U \\
&readRequestAuthorized(principal, selectors) \\
&= \forall o \in O_{all} : objectSelected(selectors, o) \Rightarrow canRead(principal, o)
\end{aligned}
\tag{1}
$$

However, the problem with determining whether Equation 1 holds, is that the set O_{all} is infinite. However, we can solve the problem using a SMT solver, by applying the following equalities:

$$
\begin{aligned}
&readRequestAuthorized(principal, selectors) \\
&= \forall o \in O_{all} : objectSelected(selectors, o) \Rightarrow canRead(principal, o) \\
&\equiv \forall o \in O_{all} : \neg objectSelected(selectors, o) \vee canRead(principal, o) \\
&\equiv \neg\neg(\forall o \in O_{all} : \neg objectSelected(selectors, o) \vee canRead(principal, o)) \\
&\equiv \neg(\exists o \in O_{all} : \neg(\neg objectSelected(selectors, o) \vee canRead(principal, o))) \\
&\equiv \neg(\exists o \in O_{all} : objectSelected(selectors, o) \wedge \neg canRead(principal, o))
\end{aligned}
\tag{2}
$$

$\exists o \in O_{all} : objectSelected(selectors, o) \wedge \neg canRead(principal, o)$ corresponds to a SMT satisfiability problem over n variables (called "constants" in SMT), one for every selector key. The Cedar integration will return `readRequestAuthorized(principal, selectors) = true` if the SMT solver returns `UNSAT`, otherwise false. In case the SMT solver returns `UNSAT`, the solver found a concrete example of a for the principal unauthorized object that could have been returned from the request.

To verify the safety of a given read request with selectors against policies written as predicates for concrete objects, the integration must:

1. Map the OpenAPI specification, which exposes information about what fields are selectable, into SMT-LIB compatible data types. The top-level API object type is called `APIObject` and ll its values are strings.

2. Define `.metadata.labels` in SMT-LIB as a map from a string to `Some(String)` or `None`. SMT-LIB supports multiple datatype variants, so this works.

3. Convert field and label selectors sent with the SubjectAccessReview into an equivalent `requestSelectsObject(o)` formula, that evaluates to true if the object would be selected.

4. The residual expression returned from the initial evaluation process is known to only reference variables scoped under `resource.stored`, as all other data was known during evaluation. The residual expression is turned into an equivalent SMT-LIB formula `principalCanRead(o)`. Only field-selectable fields can be targeted. Only Cedar operators =, ≠, `Set.contains`, `Set.containsAll`, `Set.containsAny`, `has`, `Labels.hasTag`, and `Labels.hasTag` are supported. Integers and booleans are converted into strings.

After the rewrites of the functions from high-level, generic ones, into concrete, request-scoped `requestSelectsObject(o)` and `principalCanRead(o)`, the following is asserted to the SMT solver:

```
(declare-const o APIObject)
(assert (and
  (requestSelectsObject o)
  (not (principalCanRead o))
))
```

With this approach, it is possible to use the same syntax for both reads and writes. The alternative would be to not expose `resource.stored` object at all for reads, but another selector-focused API specifically for reads. However, that would offload the cognitive complexity of reasoning about the difference of Kubernetes internals, of what is the difference between the authorization and admission stage. However, even with the unified approach, there are limitations. As discussed in the beginning of this section, only specific fields are selectable [68]. This means that there are fields visible in the Cedar schema that are not actually selectable, which could be confusing. However, through static policy analysis using the policy slicing [106] feature, the thesis project would be able to catch this error before it would be pushed to production. In similar spirit, use of operators other than the allowlisted ones should yield an error already in the development/lint stage, and this is possible to find by statically analyzing the policy.

One limitation to note with this approach currently, is that `resource.stored` is not available for `get` actions, as selectors are not currently supported for Kubernetes get requests.

5.7 Multi-cluster support

Existing Kubernetes authorization and admission control are scoped to one cluster. However, some access control policies naturally span across Kubernetes clusters, or across a Kubernetes cluster and some other system. The first requirement for federated policies to be achievable is consistent authentication across clusters, for which OIDC and/or SPIFFE are viable solutions. As future work, one could investigate if and how Cedar can function as an access control "lingua franca" language. If such an analyzable language existed in practice, heterogeneous systems could exchange information about the access control state, and thus integrate deeper. In any case could it be powerful to have a capability of expressing a desired access control policy once, and have it propagated across Kubernetes clusters and/or various systems.

This is not impossible today, however. As pointed out in a KubeCon talk Jimmy Zelinskie and I gave [13], one can layer federated access control on top of Kubernetes existing access control mechanisms already today. For example, applying a global policy across multiple clusters can be as easy as setting up a GitOps repository, and syncing to multiple targets. A controller could delete time-bound roles or bindings when they expired, for just in time authorization. More examples are enumerated in Figure 13.

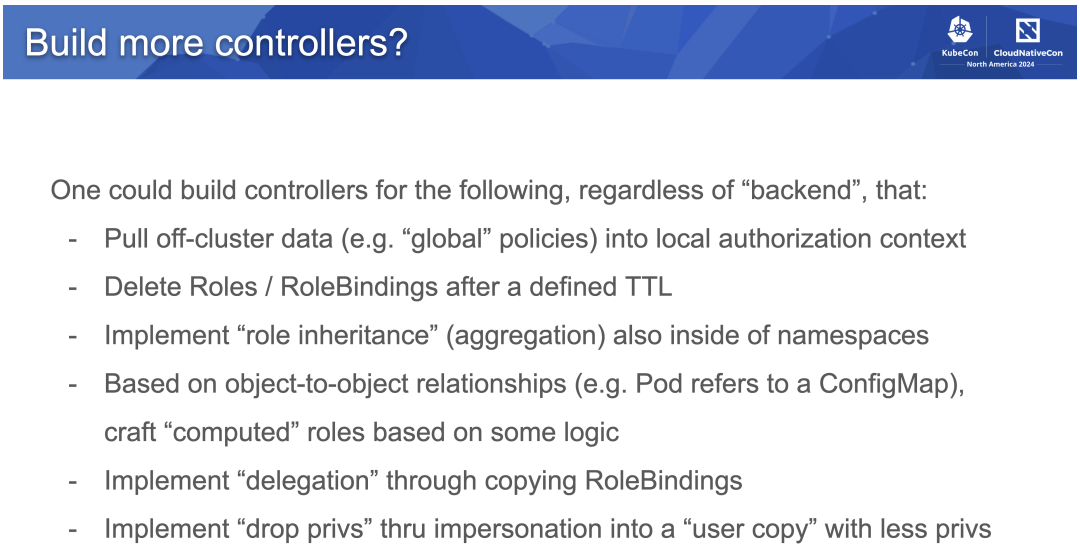


Figure 13: Slide from my and Jimmy Zelinskie’s KubeCon talk, where we discuss how additional access control capabilities for Kubernetes can be built on top of Kubernetes, without the need to be built-in. Image source: [13]

Interestingly, I have not seen clear patterns emerge in this area yet, at least not in the cloud native ecosystem. The question is then whether those that implement it keep it to themselves, if every such solution is so coupled to its environment that it does not make sense to share, or it it not easy enough to achieve. It could be a mix as well. With Cedar, a centralized policy store serving multiple clusters could be implemented, by attaching policies at various levels in the DAG. However, there is still an availability concern, as if there would be a network disconnection between the centralized policy store (wherever it is), and the clusters, the cluster could become fully unavailable.

Thus, most likely there is room for a syncing component designed for authorization data, regardless of if the access control policy is encoded as a Cedar policy or a Kubernetes RBAC Role. If such a syncing component is made, most likely the analyzability properties of Cedar, making it possible to know what entity data needs to be fetched for various enforcement points, prove useful. Also partial evaluation might make it easier to fail fast at an earlier stage in the request chain, and thus get more flexibility of when to authorize.

5.8 Policy analysis

As the number of access control policies grow, so does the maintenance cost of ensuring their correctness and reconciling them with reality. In fact, this is the core premise of why Kubernetes exists: managing containers at a large scale is a very different problem than at small scale. Kubernetes uses periodic reconciliation to make sure the actual state of the system is the desired one. Access control policies are usually either human-authored and long-lived, or generated from data and thus shorter-lived. The shorter-lived ones certainly benefit from frequent reconciliation, to make sure they match the data they were tailored to, but also longer-lived ones can be interesting to reconcile. So how does a system understand the essence of what the human policy author *actually wanted* the policy to do? One way would be to compare the policy's net effect in relation to other policies. Just as for Kubernetes, reasoning about a handful of policies is a different problem than a lot.

If Cedar policies are reduced to the common SMT-LIB format [95] [9], one may compare the relative size of two policies [101]. Here the fact that Cedar is typed is very useful: the first thing to do is to split the analysis into one sub-analysis for each *request environment* as discussed in Section 4.3.2. A request environment is a combination of a principal, action, and resource type. Knowing fully the types of the environment, the analysis becomes easier. Furthermore, it is more understandable for a human to reason about each request environment separately, both when writing a single policy, and when comparing two to each other.

Comparing two policies forms a partial order. Two policies P_1 and P_2 might be equal, one strictly larger than the other, or incomparable (if P_1 allows something that P_2 does not, and vice versa) [100]. This partial order can be useful to catch mistakes, as where one permit policy is strictly larger ("shadows") another, or when a forbid policy shadows a permit one, rendering the permit one without effect. In addition, Cedar already, to some degree, warns if a policy does nothing at all, that is, always reduces to false [9].

Putting these analysis tools in the hands of policy authors, as they write the policies, could help reduce the cognitive burden, especially as the number of policies grow, and automated systems could warn if there are policies that seem logically inconsistent. The canonical use-case for wanting certainty that a policy set is equivalent to another, is when refactoring it. Having the courage to change policy is critical for business agility, but usually the risk (and risk aversion of the person doing it) grows with policy set size. Therefore, being able to mathematically prove they are similar, or get counter-examples of when they are not, allows the policy author to write them fast with confidence.

5.9 Authorization State Queries

In addition to analyzing policies, a policy author might want to test the effect of the policy out against either mocked or live real data. In addition, an auditor is most likely interested to know who has access to sensitive resources. This process can be reactive, but automated; an automated controller can query the authorization state for a given resource from time to time, and alert an auditor if it has changed. Or better yet, proactive, by blocking such policy updates that would give a new principal access to a very sensitive resource. For these use-cases to work, the authorization engine should support answering the following questions:

1. What are the possible principal-action-resource combinations?
2. What resources of type T can principal P perform action A on?
3. What principals of type T can perform action A on resource R?

In Cedar, the answer to question (1) follows directly from the typed schema. When the answer to (1) is known, then the big question of "who has access to what?" can be broken into multiple smaller, typed questions, for each Cedar *request environment*. Questions (2) and (3) can be answered through typed partial evaluation, which for (2) returns a residual expression for a resource, and for (3) a residual expression for a principal. The residual expression forms a predicate to filter based on. In the worst case, every principal and/or resource would have to be matched against the predicate filter one-by-one client-side. It is slightly better if the API server supports this querying natively, so the auditor does not need full list access. In the best case, the server can turn the predicate into SQL or similar, and push down the filtering complexity to the backing database [113]. For some use-cases, just being able to return the predicate itself, without the filtered list can also be useful. However, as with all introspection APIs, there is a balance between a legitimate user trying to debug their access, and an attacker that want to quickly know all the things they can attack.

5.10 Multi-Level Security

Today, Kubernetes RBAC is purely additive, for good reason. Deny policies are hard to implement, reason about, and use correctly. If deny policies are supported,

meta-permissions become a pivotal design decision: who has access to deny someone else access? Most likely some hierarchy of policy tiers is then desired. However, deny policies can be efficient and understandable if there is a clear philosophy behind the authorization paradigm.

Multi-level security (MLS), as covered in Section 2.4.1, is an example of one such paradigm. One could mark API objects in Kubernetes with sensitivity labels, and enforce that even though a principal *would* have access to a resource due to some authorization policy, the principal must *as well* have a powerful enough clearance label.

MLS is of course not free of limitations and drawbacks, though. One example of what makes it hard for Kubernetes to implement deny policies in core, is the heterogeneity of Kubernetes policy author personas. It is hard to find a paradigm general enough to be useful to policy authors, without being too prone to misconfiguration or accidental complexity.

As covered in Section 3.5, Kubernetes does not have a clear DAC-like ownership model of API objects. This makes "ownership-based" authorization hard to implement, for example, for Kubernetes Secrets [78] [6]. It is technically possible to write a custom webhook authorizer that denies, but this is not supported in most cloud-provider hosted Kubernetes offerings, and too complex for most to do in practice.

However, especially for the generic control plane use-case, deny policies might be useful in practice. A generic control plane might have a tighter set of personas or more opinionated set of features, which might make it feasible to enforce some kind of deny-policy based paradigm, such as MLS.

Cedar provides the primitive of `forbid` policies, which is a good start, and default semantics such as *forbid trumps permit*. In addition, Cedar allows for multiple "policy attachment points" using the relation-based features of entities forming a directed acyclic graph (DAG), which could be used to form an order in which deny policies are respected (this requires a bit of engineering on top, however). One detail that is critical to get right, however, is making sure that a lookup of the form "can principal P perform any action on resource R?" actually verifies that the answer is "yes" for every action in all applicable request environments. It is not enough to make some "parent action", e.g. `Action::"*"` of say, concrete actions `Action::"create"` and `Action::"update"`, and just check whether `Action::"*"` is allowed. This is because a forbid policy against `Action::"update"` would not be respected in that case.

In summary, much work has to be done in this specific are still, before we have a generally-applicable deny policy framework that is usable with Kubernetes.

6 Discussion

In summary, Chapter 5 proposes answers to all three research questions. However, there are details that are tricky to get right, features required to be stabilized in Cedar, and limitations of the features.

6.1 Research Question 1: Encoding into general purpose authorization language

Section 5.4 answers the first research question (**RQ1**) by describing how Kubernetes API surface can be mapped into the Cedar data model. The main challenge encountered there was whether to "hide" Kubernetes inconsistencies or not. A policy author new to Kubernetes most likely benefits from not having to learn about accidental inconsistencies Kubernetes produced in the stable API. However, a Kubernetes expert who knows all the internals, might be confused by the Cedar interface not being a complete one-to-one mapping. Three examples are covered next.

First, the core API group in Kubernetes is signified by an empty string. An API group is turned into a *Cedar namespace*, which must be a non-empty string[19]. For now, the integration chose to put the core Kubernetes API group "" in the Cedar namespace called `core`.

Second, connectable subresources are exposed through "normal" metadata verbs. The policy author's intent when authorizing the `get` action to a set of resources, is most likely to give the bound principal the ability to read *metadata* (that is, API objects). However, authorizing `get` on a connectable subresource, might make the principal able to access the *actual data* managed by the API object, which can be unintended. For example, `get core pods`[20] in Kubernetes gives the principal access to read metadata about a workload, but `get core pods/exec` allows the principal to access and change the workload. A wildcard policy like `get core *` thus gives both metadata *and* actual data access to a principal, unless the policy is aware of this. The thesis project maps all connectable subresources to the dedidicated `connect` action instead, such that `get core *` only authorizes metadata access.

Thirdly, a principal can only impersonate another, if the impersonating principal has the authorization to do so. However, there is an inconsistency in what authorization is required, depending on what UserInfo property is impersonated. In order to impersonate a username, the impersonating principal needs `impersonate core users` access, but in order to impersonate a UID, `impersonate authentication.k8s.io users` access is needed. The thesis project chose to map all impersonation targets to the `authentication.k8s.io` API group, for consistency.

[19]There is, in fact a "root" Cedar namespace as well, but that is special. It is desired from the integration's point of view that every Kubernetes API groups would be in their own, non-root Cedar namespaces, in order to be consistent.

[20]Expressed as: <action> <api group> <resource> or <action> <api group> <resource>/<subresource>

6.2 Research Question 2: Solving long-unresolved feature requests

Sections 5.5 and 5.6 describe how the Cedar encoding helps to solve long-standing feature requests in Kubernetes. This answers the second research question (**RQ2**). However, feedback from policy authors is needed, in order to determine how useful the interface is to them. Two other limitations will be discussed here.

Firstly, one of the main concerns on the original feature request issue [6] was that policies which target labels and fields could be very easy to misconfigure by accident, rendering more harm than utility to the policy author. For example, it would be unexpected to write a policy such that a principal can edit an object they cannot read, such that after the edit, they can read it. There are a couple of options to explore as future work, as follows. For example, the API author or administrator explicitly enumerates labels and fields that can be utilized by policy authors. The system could then enforce immutability of those labels and fields. Alternatively does the Cedar interface exposed to policy authors not include the ability to target labels and fields directly. Instead could that only be done through some higher-level definition, API or function, that makes sure certain safety invariants are guaranteed. Finally, future work could investigate whether it is possible to analyze a policy set, and determine the semantic safety of it in a Kubernetes context. This follows the spirit of the Rust compiler, which warns or errors if the user tries to do something that is not considered best practice or safe.

Secondly, the set of fields which can be targeted for a read authorization policy is always a subset of what can be targeted in a write authorization policy [68]. This asymmetry is due to only certain fields being selectable for reads. This is good, as authorization policies should be kept simple. However, the asymmetry might be confusing to policy authors. One way to solve this would be to only make the field-selectable subset available for both reads and writes, but this might be too limiting for legitimate use-cases. Another way would be to use a mutating admission controller to create "virtual" field selectors, by mirroring the field value into a label, for fields that are not natively selectable. That is not a very clean method, but could work, if investigated deeper. Regardless, a Kubernetes-specific policy validator will most likely be needed, which validates whether valid Cedar policies are semantically valid Kubernetes Cedar policies, catching cases like these.

6.3 Research Question 3: Improve and unify policy author experience

Sections 5.4 and 5.8 in particular, describe how the policy author experience can be improved. Sections 5.4, 5.5 and 5.6 describe how the policy author experience could be made more uniform. These sections answer the third and final research question (**RQ3**). The VS Code experience highlighted in Figure 9 most certainly helps the policy author in their authoring of access control policies. Policy analysis might help refactoring, and understanding the policies. However, there are a couple of limitations to the approaches

Firstly, a Cedar permit policy corresponds to a Kubernetes RBAC *rule* (as given in Figure 4), not *role*. Existing Kubernetes RBAC Roles can be converted into Cedar, by naively creating $r \cdot \sum_i^n |b_i|$ Cedar permit policies from an RBAC Role, with r rules and n subject sets b_i for every relevant role binding. Cedar policy templates [114] could help de-duplicate the Cedar permit policies. Most likely, a policy author would expect some native role-based primitive similar to Kubernetes RBAC, for the Cedar integration as well.

Secondly, policy analysis might be a good tool to detect logical inconsistencies in the policies. However, if most or all of the policies are automatically generated from some more high-level API (like Kubernetes RBAC), it remains to be seen how useful the feature will be in practice. Most likely, it needs some semantic tuning to the context in order to make it user-friendly. Notable is that there was not enough time during the thesis project to actually implement privilege escalation prevention (yet), like Kubernetes RBAC does [77]. However, this can be done by enforcing that a created or updated policy permissions represents a subset of the principal's existing permissions.

Thirdly, today Kubernetes uses CEL widely across its API surface. CEL could also technically be used for conditional authorization. However, CEL does allow loops, and is thus not SMT-analyzable in a decidable way. Technically, though, if the CEL syntax used for authorization was restricted, CEL could be a great way to provide Kubernetes policy authors that already know CEL with a uniform experience, instead of forcing the policy authors to learn a new language syntax. What is interesting, however, is that Cedar and CEL are otherwise very similar, only parts of the syntax are slightly different. This means that even though CEL is used as a "frontend" in Kubernetes conditional authorization in the future, one could (and probably should) restrict it such that those CEL expressions are mappable into Cedar, and thus transitively mappable into SMT logic. Thus, Cedar could be used as a unified analysis layer, a "lingua franca" for Kubernetes authorization. Cedar, through its carefully selected features, could inform CEL usage in Kubernetes authorization which features are suitable, and which features are not.

Finally, in similar vain, there have been discussions on how label- and field selectors could be unified. Today, label selectors are more expressive than field selectors. In addition, field selectors do not necessarily map to the JSONPath of the field (an oversight at the time), or necessarily a required field. Kubernetes developers [115] discusses these issues, and how a CEL expression subset might be useful here as well. Especially useful would it be to make sure that this subset is translatable into efficient SQL [68]. Again, as Cedar already is mostly a restricted version of CEL, it might make sense to start from the Cedar AST.

6.4 Is it worth the complexity?

Engineering decisions are all about tradeoffs. Clever engineering can centralize complexity in some place, for the benefit of keeping another part simpler. That Kubernetes does not handle the complexity of, for example, selector-based conditional authorization today, means that users will have to either redesign their approach to

authorization (which can be good), implement their custom solution on top (which can be acceptable), or just not follow a least-privilege approach. If Kubernetes would take on this complexity, for the benefit of its users, there need to be a balance between impact (for users) and cost (to Kubernetes). One way to potentially minimize the cost, is to outsource part of it to another project, with the focus of handling that exact complexity. CEL has been successful in reducing the cost for Kubernetes to handle custom predicate functions. Cedar, or a mappable subset of CEL, could be successful in hiding some of the complexities that policy authors deal with today, such as conditional authorization. An integration like this might be a win to both projects: Kubernetes can outsource some internal complexity to Cedar, but in order to be able to do so, contribute to Cedar development.

Introducing more granularity is not always positive. With every added feature, there is more for policy authors to comprehend. Humans generally struggle with high dimensionality. Thus, emphasis should be placed on good tooling, and identifying a set of best practices, before more fine-grained access control is introduced. Tools that identify common mistakes are needed. An analogy here can be found from the container ecosystem. The low-level technology to isolate workloads from each other on a Linux kernel basis had been available for a long time before the industry started using containers. Containers became an industry standard first when Docker emerged, with high-quality and user-friendly tooling, and a set of best practices. I hope that we will see something similar for authorization one day. Sometimes it is not a question of whether expressing a policy is technically possible, but whether it is practically possible, with the limited time and resources policy authors have.

Finally, analyzable policies will never be expressive enough for all use-cases, by definition. However, if analyzable policies would account for 80 percent or more of the policies, that is a success. Then computers can automatically reason about the majority of the policies, and humans can take a manual look at the non-analyzable ones. An analogy here can be found from the Rust programming language, again. The idea is that the borrow checker will help the programmer catch and correct most of the runtime bugs, before ever running the program. But when the programmer needs to go beyond what expressible subject to the borrow checker's rules, the `unsafe` block can be used, which encapsulates the non-analyzable part. Hopefully, these same ideas can apply to authorization.

7 Conclusion

This thesis aims to contribute to improving Kubernetes access control. It does so by first aggregating the context the reader needs to understand the current state and problem statement. A search for suitable open source authorization engines is conducted. Then, the thesis proposes novel improvements through utilizing Cedar in a Kubernetes context. In particular, the following technical enhancements are introduced:

1. Use of a Cedar schema which is oriented around group-version-resource Cedar entities, which maps the Kubernetes API surface one-to-one and allows handling subresources.

2. Use of Cedar tags to model open records and Kubernetes map-typed lists.

3. Inclusion of request options through HTTP query parameters in the Cedar context, and query parameters for connect subresources as the payload.

4. An interface to write a single policy for both authorization and admission control, through usage of Cedar's partial evaluation feature.

5. A method to use the unified and object-scoped interface as described in (2) also for read requests, even though the read request corresponds to an unbounded amount of objects. This is achieved by partial evaluation, and mapping the selector and residual into SMT logic.

In addition, during the thesis process, I proposed two Cedar contributions [11], [12] that were accepted into the project. In addition, I presented about topics relating to the thesis at one conference in Salt Lake City [13], and one in London [14]. However, there is no production-ready end-to-end demonstration of all these features at the time of writing, as some required features are not yet fully stabilized in Cedar. However, the area is developing quickly, and I am optimistic that an end-to-end demonstration can be showcased in open source in a few months from now.

The thesis and the related project work generally manages to answer the research questions. RQ1: Kubernetes API surface can be modelled in the Cedar Policy language. RQ2: The Cedar encoding can aid adding support for conditional authorization. RQ3: The user experience can be improved by for example autocompletion, static policy validation, and policy comparison. Limitations and alternatives considered are discussed in Chapter 6. The related thesis presentation slides can be found at [76], as fully visual walkthrough of the thesis content.

I am optimistic about the outlook for the approach of managing Kubernetes access control, however, complimentary to existing solutions and designed to integrate with them. For example, Kubernetes RBAC and some ValidatingAdmissionPolicies can be translated to and analyzed in the Cedar data model. The introduction of conditional authorization will most likely help building generic platforms on top of Kubernetes, where writing custom server code is not an option. Crossplane [7] is one of many such platforms, for which the utility of the features proposed in this thesis can be evaluated.

Finally, the purpose of this project work is to collaboratively improve access control with the wider community. I look forward to continuing the good discussions with the Cedar community, as with the Kubernetes community. A practical way this work could materialize on the Kubernetes side, is through writing a Kubernetes Enhancement Proposal [102] on if and how the Kubernetes SubjectAccessReview API could be changed in a backwards-compatible way to support conditional authorization.

The thesis project is aimed to be open sourced and merged into the Cedar Access Control for Kubernetes project [10], once that project has moved to the `cedar-policy` GitHub organization.

7.1 Future Work

As with any software project, even though it is finished, there are lots of venues to keep improving. Many ideas for future improvements have been mentioned throughout the thesis text. However, a non-exhaustive list of areas for future work is here as follows:

- Showcase the end-to-end demonstration of the Kubernetes integration, once typed partial evaluation [97] is available in Cedar.
- Implement privilege escalation prevention, once Cedar Policy compilation into SMT logic [96] is available in Cedar.
- Implement policy validator that is Kubernetes-context and semantics-aware, and that warns or errors for valid Cedar policies, that logically wrong in a Kubernetes context, once Entity Slicing with Data Tries [106] is available in Cedar. For example, analyze conditionally authorized policies, and enforce that unexpected behavior is not allowed by the policy (e.g. being able to edit a for the editor unreadable object such that it becomes readable for the editor).
- Investigate a feature that would allow "downgrading" and "right-sizing" a list or watch request to the privileges of the principal. For example, if the principal can only list with label `owner=lucas`, let the API server apply this selector automatically, instead of denying the request.
- Investigate if and how Kubernetes could natively support conditional authorization, e.g. through finding the intersection of CEL and Cedar syntaxes, and using such analyzable expressions as the conditions in Kubernetes' SubjectAccessReview.
- Align the development of a new selector type [115] in Kubernetes with the expressiveness of Cedar. Most likely the intersection between CEL and Cedar could be a helpful start.
- Investigate what is the most natural way to expose a RBAC interface using Cedar in a Kubernetes context, possibly using Cedar policy templates [114].

- Investigate the feasibility of integrating deny policies into Kubernetes through Cedar, such that e.g. Multi-level Security-like policies could be expressed. Investigate what organization of "policy tiers" is the most intuitive to Kubernetes policy authors.
- Investigate how a policy author could in the most straightforward way express the intent that only a given set of allowlisted fields can change. That is, the semantic difference of all other fields must be equal for updates containing both a new and old object. It could be some kind of "macro", which depending on the type autogenerates `resource.stored.v1.fieldA == resource.request.v1.fieldA && resource.stored.v1.fieldB == resource.request.v1.fieldB && ...` statements would suffice?
- Investigate how compound or composite authorization would work in the most natural way with Cedar. Most likely, this feature would be built on top of all of the other features mentioned so far.
- Investigate how multi-cluster authorization scenarios could be easier to handle, through policy-format-agnostic reconcilers of policies.
- Investigate if and how Cedar could be used to analyze "inherited" permissions, that a principal has implicitly through a Kubernetes controller.
- Investigate if Cedar can be used as a "lingua franca" of authorization, which through its analysis capabilities allows different CNCF projects reason about each others' permission models in a deeper way than before.

References

[1] Amazon Web Services, Inc., *What is Cloud Computing? - Cloud Computing Services, Benefits, and Types - AWS*, en-US, 2025. [Online]. Available: https://aws.amazon.com/what-is-cloud-computing/ (visited on 05/24/2025).

[2] The Kubernetes Authors, *Kubernetes API Concepts*, en, Section: docs, 2024. [Online]. Available: https://kubernetes.io/docs/reference/using-api/api-concepts/ (visited on 02/03/2025).

[3] The OWASP Foundation, *OWASP API Security Project*, en, 2023. [Online]. Available: https://owasp.org/www-project-api-security/ (visited on 05/23/2025).

[4] Cloud Native Computing Foundation, *New SlashData report: 5.6 million developers use Kubernetes, an increase of 67% over one year*, en-US, Dec. 2021. [Online]. Available: https://www.cncf.io/blog/2021/12/20/new-slashdata-report-5-6-million-developers-use-kubernetes-an-increase-of-67-over-one-year/ (visited on 05/24/2025).

[5] The Kubernetes Authors, *Sig-security/sig-security-external-audit/security-audit-2021-2022/findings/Kubernetes v1.24 Final Report.pdf at main · kubernetes/sig-security*, en, 2023. [Online]. Available: https://github.com/kubernetes/sig-security/blob/main/sig-security-external-audit/security-audit-2021-2022/findings/Kubernetes%20v1.24%20Final%20Report.pdf (visited on 05/24/2025).

[6] The Kubernetes Authors, *Support label selector in RBAC · Issue #44703 · kubernetes/kubernetes*, 2017. [Online]. Available: https://github.com/kubernetes/kubernetes/issues/44703#issuecomment-324826356 (visited on 05/15/2025).

[7] C. Community, *Crossplane Docs · v1.17 · Crossplane Introduction*, en, 2024. [Online]. Available: https://docs.crossplane.io/v1.18/getting-started/introduction/ (visited on 11/25/2024).

[8] Google, *Google/cel-spec*, original-date: 2017-09-04T18:33:35Z, May 2025. [Online]. Available: https://github.com/google/cel-spec (visited on 05/24/2025).

[9] J. Cutler, C. Disselkoen, A. Eline, *et al.*, *Cedar: A new language for expressive, fast, safe, and analyzable authorization*, en, 2024. [Online]. Available: https://www.amazon.science/publications/cedar-a-new-language-for-expressive-fast-safe-and-analyzable-authorization (visited on 03/11/2025).

[10] M. Hausler, *Awslabs/cedar-access-control-for-k8s*, original-date: 2024-10-23T18:50:08Z, May 2025. [Online]. Available: https://github.com/awslabs/cedar-access-control-for-k8s (visited on 05/17/2025).

[11] L. Käldström, *Allow reauthorization with also initially undiscovered unknowns in the Expr #1466 - cedar-policy/cedar*, original-date: 2023-04-25T15:13:59Z, May 2025. [Online]. Available: https://github.com/cedar-policy/cedar/pull/1466 (visited on 05/16/2025).

[12] L. Käldström, *Pretty-print the Cedarschema files using two spaces by luxas · Pull Request #1587 · cedar-policy/cedar*, May 2025. [Online]. Available: https://github.com/cedar-policy/cedar/pull/1587 (visited on 05/17/2025).

[13] *Expanding the Capabilities of Kubernetes Access Control - Jimmy Zelinskie & Lucas Käldström*, Nov. 2024. [Online]. Available: https://www.youtube.com/watch?v=IXHCSSQeXBg (visited on 05/18/2025).

[14] L. Käldström and M. Hausler, *End to End Message Authenticity in Cloud Native Systems*, en, 2025. [Online]. Available: https://speakerdeck.com/luxas/end-to-end-message-authenticity-in-cloud-native-systems (visited on 05/23/2025).

[15] The SPIFFE Authors, *SPIFFE Federation*, en, 2022. [Online]. Available: https://github.com/spiffe/spiffe/blob/main/standards/SPIFFE_Federation.md (visited on 11/26/2024).

[16] The SPIFFE Authors, *The SPIFFE Identity and Verifiable Identity Document*, en, 2022. [Online]. Available: https://github.com/spiffe/spiffe/blob/main/standards/SPIFFE-ID.md (visited on 11/25/2024).

[17] International Telecommunication Union, *X.509 : Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks*, 2019. [Online]. Available: https://www.itu.int/rec/T-REC-X.509 (visited on 05/22/2025).

[18] Cloudflare Inc., *What is mTLS?*, en-us, 2024. [Online]. Available: https://www.cloudflare.com/learning/access-management/what-is-mutual-tls/ (visited on 11/26/2024).

[19] S. Boeyen, S. Santesson, T. Polk, R. Housley, S. Farrell, and D. Cooper, *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, Request for Comments, Num Pages: 151, May 2008. DOI: 10.17487/RFC5280. [Online]. Available: https://datatracker.ietf.org/doc/rfc5280 (visited on 11/26/2024).

[20] A. Langley, *ImperialViolet - No, don't enable revocation checking*, 2014. [Online]. Available: https://www.imperialviolet.org/2014/04/19/revchecking.html (visited on 11/26/2024).

[21] OpenID Foundation, *How OpenID Connect Works*, en-US, Feb. 2023. [Online]. Available: https://openid.net/developers/how-connect-works/ (visited on 11/26/2024).

[22] M. B. Jones, J. Bradley, and N. Sakimura, *JSON Web Token (JWT)*, Request for Comments, Num Pages: 30, May 2015. DOI: 10.17487/RFC7519. [Online]. Available: https://datatracker.ietf.org/doc/rfc7519 (visited on 11/26/2024).

[23] D. Hardt, *The OAuth 2.0 Authorization Framework*, Request for Comments, Num Pages: 76, Oct. 2012. DOI: 10.17487/RFC6749. [Online]. Available: https://datatracker.ietf.org/doc/rfc6749 (visited on 11/26/2024).

[24] N. Sakimura, J. Bradley, M. Jones, and E. Jay, *OpenID Connect Discovery 1.0 incorporating errata set 2*, 2023. [Online]. Available: https://openid.net/specs/openid-connect-discovery-1_0.html (visited on 11/26/2024).

[25] The SPIFFE Authors, *The JWT SPIFFE Verifiable Identity Document*, en, 2022. [Online]. Available: https://github.com/spiffe/spiffe/blob/main/standards/JWT-SVID.md (visited on 11/26/2024).

[26] Amazon Web Services, Inc., *OIDC federation - AWS Identity and Access Management*, 2025. [Online]. Available: https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_providers_oidc.html (visited on 05/23/2025).

[27] A. Backman, J. Richer, and M. Sporny, *HTTP Message Signatures*, Request for Comments, Num Pages: 95, Feb. 2024. DOI: 10.17487/RFC9421. [Online]. Available: https://datatracker.ietf.org/doc/rfc9421 (visited on 11/26/2024).

[28] D. Fett, B. Campbell, J. Bradley, T. Lodderstedt, M. B. Jones, and D. Waite, "OAuth 2.0 Demonstrating Proof of Possession (DPoP)", Internet Engineering Task Force, Request for Comments RFC 9449, Sep. 2023, Num Pages: 39. DOI: 10.17487/RFC9449. [Online]. Available: https://datatracker.ietf.org/doc/rfc9449 (visited on 05/23/2025).

[29] E. Heilman, L. Mugnier, A. Filippidis, *et al.*, *OpenPubkey: Augmenting OpenID Connect with User held Signing Keys*, Publication info: Preprint., 2023. [Online]. Available: https://eprint.iacr.org/2023/296 (visited on 05/23/2025).

[30] The SPIFFE Authors, *SPIFFE.md at main · spiffe/spiffe*, en, 2022. [Online]. Available: https://github.com/spiffe/spiffe/blob/main/standards/SPIFFE.md (visited on 05/23/2025).

[31] The SPIFFE Authors, *The SPIFFE Workload API*, en, 2022. [Online]. Available: https://github.com/spiffe/spiffe/blob/main/standards/SPIFFE_Workload_API.md (visited on 11/26/2024).

[32] J. Beda, *Who's Calling? Production Identity in a Microservices World*, 2016. [Online]. Available: http://slides.eightypercent.net/spiffe-intro/index.html#1 (visited on 11/26/2024).

[33] C. Box, A. Glick, and A. Jessup, *Kubernetes Podcast from Google: Episode 45 - SPIFFE, with Andrew Jessup*, en-US, 2019. [Online]. Available: https://kubernetespodcast.com/episode/045-spiffe/ (visited on 05/23/2025).

[34] Cloud Native Computing Foundation, *Cloud Native Computing Foundation*, en-US, 2024. [Online]. Available: https://www.cncf.io/ (visited on 11/25/2024).

[35] The SPIFFE Authors, *SPIFFE_federation.md at main · spiffe/spiffe · GitHub*, 2022. [Online]. Available: https://github.com/spiffe/spiffe/blob/main/standards/SPIFFE_Federation.md (visited on 05/23/2025).

[36] A. Jøsang, A Consistent Definition of Authorization, en, in *Security and Trust Management*, Cham: Springer International Publishing, 2017, pp. 134–144, ISBN: 978-3-319-68063-7. DOI: 10.1007/978-3-319-68063-7_9.

[37] B. Y. Fraser, *Site Security Handbook*, Request for Comments, Num Pages: 75, Sep. 1997. DOI: 10.17487/RFC2196. [Online]. Available: https://datatracker.ietf.org/doc/rfc2196 (visited on 02/03/2025).

[38] Merriam-Webster, Incorporated, *Authorize definition*, en, Jan. 2025. [Online]. Available: https://www.merriam-webster.com/dictionary/authorize (visited on 02/03/2025).

[39] V. C. Hu, D. Ferraiolo, R. Kuhn, *et al.*, *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*, en, Jan. 2014. DOI: 10.6028/NIST.SP.800-162. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-162.pdf (visited on 02/03/2025).

[40] International Organization for Standardization, *ISO/IEC 27000:2018*, en, 2018. [Online]. Available: https://www.iso.org/standard/73906.html (visited on 02/03/2025).

[41] The Cedar Authors, *How Cedar Authorization Works*, en-US, May 2025. [Online]. Available: https://docs.cedarpolicy.com/auth/authorization.html (visited on 05/23/2025).

[42] INCITS, *INCITS 499-2018 - Information Technology - Next Generation Access Control - Functional Architecture (NGAC-FA)*, 2018. [Online]. Available: https://webstore.ansi.org/standards/incits/incits4992018 (visited on 02/03/2025).

[43] Authzed Inc., *JWT Authorization: Avoiding Common Pitfalls*, en, 2023. [Online]. Available: https://authzed.com/blog/pitfalls-of-jwt-authorization (visited on 11/26/2024).

[44] V. Hu, D. Ferraiolo, R. Chandramouli, and D. Kuhn, *Attribute-Based Access Control*, en-US, 2017. [Online]. Available: https://ieeexplore.ieee.org/document/9100400 (visited on 02/03/2025).

[45] OpenID AuthZEN, *Authorization API 1.0 – draft 03*, 2025. [Online]. Available: https://openid.github.io/authzen/.

[46] B. W. Lampson, "Protection", *Proc. Fifth Princeton Symposium on Information Sciences and Systems*, pp. 437–443, Mar. 1971, ISSN: 0163-5980. DOI: 10.1145/775265.775268. [Online]. Available: https://dl.acm.org/doi/10.1145/775265.775268 (visited on 02/03/2025).

[47] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems", *Commun. ACM*, vol. 19, no. 8, pp. 461–471, Aug. 1976, ISSN: 0001-0782. DOI: 10.1145/360303.360333. [Online]. Available: https://dl.acm.org/doi/10.1145/360303.360333 (visited on 02/03/2025).

[48] M. Davis, *Computability and Unsolvability*. 1958, ISBN: 978-0-486-61471-7.

[49] D. Ferraiolo and R. Kuhn, "Role-Based Access Controls", en, in *Proceedings of the 15th National Computer Security Conference*, Conference Name: 15th National Computer Security Conference (NCSC); 10/13/1992 - 10/16/1992; Baltimore, Maryland, United States, National Institute of Standards and Technoloty, Oct. 1992, pp. 554–563. DOI: 10/13/rolebased-access-controls/final. [Online]. Available: https://csrc.nist.gov/pubs/conference/1992/10/13/rolebased-access-controls/final (visited on 05/23/2025).

[50] R. Pang, R. Cáceres, M. Burrows, *et al.*, Zanzibar: Google's consistent, global authorization system, in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19, USA: USENIX Association, Jul. 2019, pp. 33–46, ISBN: 978-1-939133-03-8. [Online]. Available: https://research.google/pubs/zanzibar-googles-consistent-global-authorization-system/ (visited on 02/03/2025).

[51] Authzed Inc., *SpiceDB*, en, 2025. [Online]. Available: https://authzed.com/spicedb (visited on 05/23/2025).

[52] The OpenFGA Authors, *Fine-Grained Authorization | OpenFGA*, en, 2025. [Online]. Available: https://openfga.dev/ (visited on 05/23/2025).

[53] Authzed Inc., *Google-Scale Authorization: Getting to 1 Million QPS on SpiceDB Dedicated with CockroachDB*, en, 2024. [Online]. Available: https://authzed.com/blog/google-scale-authorization (visited on 05/23/2025).

[54] M. Burgess, *In Search of Certainty: Ruling The Machines That Rule The World*. O'Reilly Media, Inc., 2015, ISBN: 978-1-4919-2336-8. [Online]. Available: http://markburgess.org/certainty.html (visited on 11/25/2024).

[55] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, Large-scale cluster management at Google with Borg, en, in *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015. [Online]. Available: https://research.google/pubs/pub43438.pdf (visited on 11/25/2024).

[56] L. Käldström, "Encoding human-like operational knowledge using declarative Kubernetes operator patterns", en, Bachelor's thesis, Aalto University, 2021. [Online]. Available: https://github.com/luxas/research/blob/main/bsc_thesis.pdf (visited on 11/25/2024).

[57] Google LLC, *AIP-111: Planes*, 2023. [Online]. Available: https://google.aip.dev/111 (visited on 11/25/2024).

[58] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures", en, Doctoral Dissertation, University of California, Irvine, 2000.

[59] Tim Hockin, *Kubernetes Controllers - are they loops or events?*, en, 2021. [Online]. Available: https://speakerdeck.com/thockin/kubernetes-controllers-are-they-loops-or-events (visited on 02/03/2025).

[60] Honeypot, *Kubernetes: The Documentary*, Jan. 2022. [Online]. Available: https://www.youtube.com/watch?v=BE77h7dmoQU (visited on 11/25/2024).

[61] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, Omega: flexible, scalable schedulers for large compute clusters, in *SIGOPS European Conference on Computer Systems (EuroSys)*, Prague, Czech Republic, 2013, pp. 351–364. [Online]. Available: http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf (visited on 11/25/2024).

[62] Google LLC, *The Go Programming Language*, en, 2024. [Online]. Available: https://go.dev/ (visited on 11/25/2024).

[63] The etcd Authors, *Etcd*, en, 2024. [Online]. Available: https://etcd.io/ (visited on 10/11/2021).

[64] The OpenAPI Initiative, *OpenAPI Specification v3.1.0*, 2021. [Online]. Available: https://spec.openapis.org/oas/v3.1.0.html (visited on 05/17/2025).

[65] The Kubernetes Authors, *Custom Resources*, en, Section: docs, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/ (visited on 11/25/2024).

[66] The Kubernetes Authors, *API Conventions - community/contributors/devel/sig-architecture/api-conventions.md at master · kubernetes/community*, en, 2023. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md (visited on 05/17/2025).

[67] S. Gössner, G. Normington, and C. Bormann, "JSONPath: Query Expressions for JSON", Internet Engineering Task Force, Request for Comments RFC 9535, Feb. 2024, Num Pages: 62. DOI: 10.17487/RFC9535. [Online]. Available: https://datatracker.ietf.org/doc/rfc9535 (visited on 05/19/2025).

[68] The Kubernetes Authors, *KEP-4358: Custom Resource Field Selectors · kubernetes/enhancements*, 2024. [Online]. Available: https://github.com/kubernetes/enhancements/tree/master/keps/sig-api-machinery/4358-custom-resource-field-selectors (visited on 05/15/2025).

[69] OpenAI, *API, ChatGPT & Sora Facing Issues*, en, Dec. 2024. [Online]. Available: https://status.openai.com/incidents/01JMYB483C404VMPCW726E8MET (visited on 05/19/2025).

[70] The Kubernetes Authors, *API Priority and Fairness*, en, Section: docs, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/flow-control/ (visited on 03/19/2025).

[71] The Kubernetes Authors, *Authenticating*, en, Section: docs, 2025. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/authentication/ (visited on 05/19/2025).

[72] Tremolo Security, *Kube-oidc-proxy*, en, 2025. [Online]. Available: https://github.com/TremoloSecurity/kube-oidc-proxy (visited on 05/19/2025).

[73] The Kubernetes Authors, *Kubernetes Authorization*, en, Section: docs, 2025. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/authorization/ (visited on 05/15/2025).

[74] The Cedar Authors, *Compound authorization*, en-US, Mar. 2025. [Online]. Available: https://docs.cedarpolicy.com/bestpractices/bp-compound-auth.html (visited on 03/19/2025).

[75] The Kubernetes Authors, *KEP-5018 - enhancements/keps/sig-auth/5018-dra-adminaccess at master · kubernetes/enhancements*, 2025. [Online]. Available: https://github.com/kubernetes/enhancements/tree/master/keps/sig-auth/5018-dra-adminaccess (visited on 05/17/2025).

[76] L. Käldström, *Presentation: Usable Access Control in Cloud Management Systems*, en, 2025. [Online]. Available: https://speakerdeck.com/luxas/usable-access-control-in-cloud-management-systems (visited on 05/25/2025).

[77] The Kubernetes Authors, *Using RBAC Authorization*, en, Section: docs, 2025. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/rbac/ (visited on 05/19/2025).

[78] The Crossplane Authors, *Crossplane/design/design-doc-external-secret-stores.md at main · crossplane/crossplane*, 2022. [Online]. Available: https://github.com/crossplane/crossplane/blob/main/design/design-doc-external-secret-stores.md (visited on 05/19/2025).

[79] The Kubernetes Authors, *[WIP] Begin KEP for fine-grained authz #3617 - kubernetes/enhancements*, original-date: 2016-05-02T18:51:30Z, 2023. [Online]. Available: https://github.com/kubernetes/enhancements (visited on 05/24/2025).

[80] Google LLC, *Protocol Buffers*, en, 2024. [Online]. Available: https://protobuf.dev/ (visited on 11/25/2024).

[81] The Kubernetes Authors, *Validating Admission Policy*, en, Section: docs, 2025. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/validating-admission-policy/ (visited on 05/26/2025).

[82] K. Hietala and E. Torlak, *Lean Into Verified Software Development | AWS Open Source Blog*, en-US, Section: Amazon Verified Permissions, Apr. 2024. [Online]. Available: https://aws.amazon.com/blogs/opensource/lean-into-verified-software-development/ (visited on 03/11/2025).

[83] T. Junttila, *CS-E3220: Propositional satisfiability and SAT solvers documentation*, 2024. [Online]. Available: https://users.aalto.fi/~tjunttil/2024-DP-AUT/notes-sat/overview.html (visited on 03/11/2025).

[84] S. A. Cook, The complexity of theorem-proving procedures, in *Proceedings of the third annual ACM symposium on Theory of computing*, ser. STOC '71, New York, NY, USA: Association for Computing Machinery, May 1971, pp. 151–158, ISBN: 978-1-4503-7464-4. DOI: 10.1145/800157.805047. [Online]. Available: https://dl.acm.org/doi/10.1145/800157.805047 (visited on 03/11/2025).

[85] A. Church, "A note on the Entscheidungsproblem", en, *The Journal of Symbolic Logic*, vol. 1, no. 1, pp. 40–41, Mar. 1936, ISSN: 0022-4812, 1943-5886. DOI: 10.2307/2269326. [Online]. Available: https://www.cambridge.org/core/journals/journal-of-symbolic-logic/article/abs/note-on-the-entscheidungsproblem/9461BEAD94BB16D56EC78933D7D67DEF#article (visited on 05/26/2025).

[86] The Cedar Authors, *RFC21 - rfcs/archive/rfc/0021-any-and-all-operators.md at main · cedar-policy/rfcs*, 2024. [Online]. Available: https://github.com/cedar-policy/rfcs/blob/main/archive/rfc/0021-any-and-all-operators.md (visited on 05/17/2025).

[87] The SMT-LIB Authors, *SMT-LIB The Satisfiability Modulo Theories Library*, 2025. [Online]. Available: https://smt-lib.org/ (visited on 03/11/2025).

[88] Microsoft Corporation, *Z3Prover/z3*, original-date: 2015-03-26T18:16:07Z, Mar. 2025. [Online]. Available: https://github.com/Z3Prover/z3 (visited on 03/11/2025).

[89] The cvc Authors, *Cvc5/cvc5*, original-date: 2012-12-06T18:45:04Z, Mar. 2025. [Online]. Available: https://github.com/cvc5/cvc5 (visited on 03/11/2025).

[90] The Cedar Authors, *Avoid mutable identifiers - Cedar Policy Language Reference Guide*, en-US, Mar. 2025. [Online]. Available: https://docs.cedarpolicy.com/bestpractices/bp-mutable-identifiers.html (visited on 03/19/2025).

[91] D. McAdams, *Why does Cedar ignore policies that error? - Cedarland Blog*, 2023. [Online]. Available: https://cedarland.blog/design/why-ignore-errors/content.html (visited on 03/19/2025).

[92] The Cedar Authors, *Design patterns - Cedar Policy Language Reference Guide*, en-US, Mar. 2025. [Online]. Available: https://docs.cedarpolicy.com/overview/patterns.html (visited on 03/19/2025).

[93] The Cedar Authors, *Resource containers - Cedar Policy Language Reference Guide*, en-US, Feb. 2025. [Online]. Available: https://docs.cedarpolicy.com/bestpractices/bp-resources-containers.html (visited on 03/11/2025).

[94] The Cedar Authors, *Separate principals & containers - Cedar Policy Language Reference Guide*, en-US, Feb. 2025. [Online]. Available: https://docs.cedarpolicy.com/bestpractices/bp-separate-principals.html (visited on 03/11/2025).

[95] A. M. Wells, *What's analyzable - Cedar blog*, 2024. [Online]. Available: https://www.cedarpolicy.com/blog/whats-analyzable (visited on 03/11/2025).

[96] The Cedar Authors, *Define roadmap for symbolic compiler/smt analyzer · Issue #1385 · cedar-policy/cedar*, en, 2024. [Online]. Available: https://github.com/cedar-policy/cedar/issues/1385 (visited on 03/19/2025).

[97] E. Torlak, *Simple type-aware partial evaluation - cedar-policy/rfcs*, Mar. 2025. [Online]. Available: https://github.com/cedar-policy/rfcs/pull/95 (visited on 03/19/2025).

[98] The Cedar Authors, *[Sandbox] Cedar Policy #371 - cncf/sandbox*, original-date: 2022-10-03T22:48:09Z, May 2025. [Online]. Available: https://github.com/cncf/sandbox (visited on 05/17/2025).

[99] The Kubernetes Authors, *NCC-E003660-DXX: Lack of Cohesion Between Core Access Control Mechanisms · Issue #118985 · kubernetes/kubernetes*, 2023. [Online]. Available: https://github.com/kubernetes/kubernetes/issues/118985 (visited on 05/26/2025).

[100] W. Eiers, G. Sankaran, A. Li, E. O'Mahony, B. Prince, and T. Bultan, Quacky: Quantitative Access Control Permissiveness Analyzer, in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22, New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 1–5, ISBN: 978-1-4503-9475-8. DOI: 10.1145/3551349.3559530. [Online]. Available: https://dl.acm.org/doi/10.1145/3551349.3559530 (visited on 05/15/2025).

[101] J. Backes, P. Bolignano, B. Cook, *et al.*, Semantic-based Automated Reasoning for AWS Access Policies using SMT, in *2018 Formal Methods in Computer Aided Design (FMCAD)*, Oct. 2018, pp. 1–9. DOI: 10.23919/FMCAD.2018.8602994. [Online]. Available: https://ieeexplore.ieee.org/document/8602994 (visited on 03/19/2025).

[102] The Kubernetes Authors, *Enhancement Tracking and Backlog - kubernetes/enhancements*, original-date: 2016-05-02T18:51:30Z, May 2025. [Online]. Available: https://github.com/kubernetes/enhancements (visited on 05/15/2025).

[103] The Kubernetes Authors, *Support for managing revoked certs · Issue #18982 · kubernetes/kubernetes*, en, 2024. [Online]. Available: https://github.com/kubernetes/kubernetes/issues/18982 (visited on 02/03/2025).

[104] T. Hoare, *Null References: The Billion Dollar Mistake*, en, Aug. 2009. [Online]. Available: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/ (visited on 05/15/2025).

[105] Microsoft Corporation, *Language Server Protocol Overview - Visual Studio*, en-us, 2023. [Online]. Available: https://learn.microsoft.com/en-us/visualstudio/extensibility/language-server-protocol?view=vs-2022 (visited on 05/15/2025).

[106] O. Flatt, *Entity Slicing using Data Tries - cedar-policy/rfcs*, Mar. 2025. [Online]. Available: https://github.com/cedar-policy/rfcs/pull/74 (visited on 03/19/2025).

[107] The Cedar Authors, *Entity Slice Validation - cedar-policy/rfcs*, en, 2024. [Online]. Available: https://github.com/cedar-policy/rfcs/tree/main/text (visited on 05/15/2025).

[108] The Cedar Authors, *RFC68 - rfcs/archive/rfc/0068-entity-tags.md at main · cedar-policy/rfcs*, 2024. [Online]. Available: https://github.com/cedar-policy/rfcs/blob/main/archive/rfc/0068-entity-tags.md (visited on 05/17/2025).

[109] The Kubernetes Authors, *CRD Processing - The Kubebuilder Book*, 2019. [Online]. Available: https://book.kubebuilder.io/reference/markers/crd-processing (visited on 05/17/2025).

[110] The Cedar Authors, *RFC82 - rfcs/text/0082-entity-tags.md at main · cedar-policy/rfcs*, 2024. [Online]. Available: https://github.com/cedar-policy/rfcs/blob/main/text/0082-entity-tags.md (visited on 05/17/2025).

[111] The Cedar Authors, *Cedar-policy/vscode-cedar*, original-date: 2023-08-01T20:37:25Z, Apr. 2025. [Online]. Available: https://github.com/cedar-policy/vscode-cedar (visited on 05/15/2025).

[112] The Kubernetes Authors, *Enhancements/keps/sig-auth/4601-authorize-with-selectors at master · kubernetes/enhancements*, 2024. [Online]. Available: https://github.com/kubernetes/enhancements/tree/master/keps/sig-auth/4601-authorize-with-selectors (visited on 05/25/2025).

[113] Amazon Web Services, Inc., *A quick guide to partial evaluation - Cedarland Blog*, 2024. [Online]. Available: https://cedarland.blog/usage/partial-evaluation/content.html (visited on 11/26/2024).

[114] The Cedar Authors, *Policy templates*, en-US, May 2025. [Online]. Available: https://docs.cedarpolicy.com/policies/templates.html (visited on 05/25/2025).

[115] The Kubernetes Authors, *Issue-#128154: Support for expanded 'operators' (e.g. In) in field selectors · kubernetes/kubernetes*, 2024. [Online]. Available: https://github.com/kubernetes/kubernetes/issues/128154 (visited on 05/15/2025).

A Cedar Policy and Schema Examples

Small example snippet of the generated schema (the real file is around 5000 lines long), as described in Section 5.3:

```
namespace k8s {
  entity User = {
    "extra": meta::StringToStringSetMap,
    "groups": Set<__cedar::String>,
    "uid": __cedar::String,
    "username": __cedar::String
  };

  entity Namespace = {
    "metadata": meta::V1ObjectMeta,
    "name": __cedar::String
  };

  entity Resource in [k8s::Namespace] = {
    "apiGroup": __cedar::String,
    "apiVersion": __cedar::String,
    "name": __cedar::String,
    "namespace"?: k8s::Namespace,
    "resourceCombined": __cedar::String
  };

  // Just a short snippet of principals and resources bound
  // to the action here, lots more in the real schema
  action "*" appliesTo {
    principal: [k8s::User],
    resource: [k8s::Resource, core::pods, core::pods_status],
    context: {}
  };

  // Just showcase one action in this snippet
  action "create" in [k8s::Action::"*"] appliesTo {
    principal: [k8s::User],
    resource: [k8s::Resource, core::pods, core::pods_status],
    context: {}
  };
}

namespace core {
  entity pods in [k8s::Namespace] = {
    "apiGroup": __cedar::String,          // always ""
```

```
    "apiVersion": __cedar::String,        // always "v1"
    "name"?: __cedar::String,
    "namespace": k8s::Namespace,
    "request"?: core::VersionedPod,
    "resourceCombined": __cedar::String,  // always "pods"
    "stored"?: core::VersionedPod
  };
  type VersionedPod = {
    "apiVersion": __cedar::String,        // always "v1"
    "kind": __cedar::String,              // always "Pod"
    "metadata": meta::V1ObjectMeta,
    "v1"?: core::V1Pod
  };
  type V1Pod = {
    "spec"?: core::V1PodSpec,
    "status"?: core::V1PodStatus
  };
  type V1PodSpec = {
    // lots of fields
  };
  type V1PodStatus = {
    // lots of fields
  };

  entity pods_status in [k8s::Namespace] = {
    "apiGroup": __cedar::String,          // always ""
    "apiVersion": __cedar::String,        // always "v1"
    "name": __cedar::String,
    "namespace": k8s::Namespace,
    "request"?: core::VersionedPodStatus,
    "resourceCombined": __cedar::String,  // always "pods/status"
    "stored"?: core::VersionedPodStatus
  };
  type VersionedPodStatus = {
    "apiVersion": __cedar::String,        // always "v1"
    "kind": __cedar::String,              // always "Pod"
    "v1": core::V1PodStatus
  };
}

namespace meta {
  entity StringToStringMap = {
    "keys": Set<__cedar::String>
  } tags __cedar::String;
```

```
  entity StringToStringSetMap = {
    "keys": Set<__cedar::String>
  } tags Set<__cedar::String>;

  type V1ObjectMeta = {
    "annotations": meta::StringToStringMap,
    "creationTimestamp": __cedar::String,
    "deletionTimestamp"?: __cedar::String,
    "finalizers": Set<__cedar::String>,
    "generateName"?: __cedar::String,
    "labels": meta::StringToStringMap,
    "resourceVersion": __cedar::String,
    "uid": __cedar::String
  };
}
```

```
// Permit Kubernetes users in the with-owner-labels group to only
// operate uniformly across Secrets, whenever the label is set.
permit (
  principal is k8s::User,
  // any action: get, list, watch, create, update, patch, deletecollection, delete
  action,
  resource is core::secrets
) when {
  principal.groups.contains("with-owner-labels") &&
  (if resource has stored then
    resource.stored.metadata.labels.hasTag("owner") &&
    resource.stored.metadata.labels.getTag("owner") == principal.username
  else true) &&
  (if resource has request then
    resource.request.metadata.labels.hasTag("owner") &&
    resource.request.metadata.labels.getTag("owner") == principal.username
  else true)
};
```

Figure A1: A shorter, more concise version of Figure 10, where the label restriction is applied whenever it is applicable for the verb, without having to split the policy into three parts.