OWSM: Empowering Rego for Stateful Access Control

Massimiliano Baldo[1], Fabio Ionut Ion[1],
Marino Miculan[1,2], Matteo Paier[1,3], and Vincenzo Riccio[1]

[1] University of Udine - Dept. of Mathematics, Computer Science and Physics, Italy.
massimiliano.baldo@uniud.it, marino.miculan@uniud.it, vincenzo.riccio@uniud.it
[2] Ca' Foscari University of Venice - Dept. of Environmental Sciences, Informatics and Statistics, Italy.
[3] IMT Alti Studi Lucca, Italy. matteo.paier@imtlucca.it

Abstract

Service mesh technologies have emerged as a powerful tool for managing microservices communication. However, enforcing complex access control policies often requires stateful mechanisms, which are not directly supported by policy languages like Rego. To address this limitation, we propose the *OPA Wrapper State Manager* (OWSM). OWSM maintains a separate state store that can be accessed during policy evaluation. This enables the specification and enforcement of stateful access control policies using Rego's declarative syntax. We evaluate the performance and overhead of OWSM through experiments, demonstrating its effectiveness in enhancing the capabilities of service mesh environments.

1 Introduction

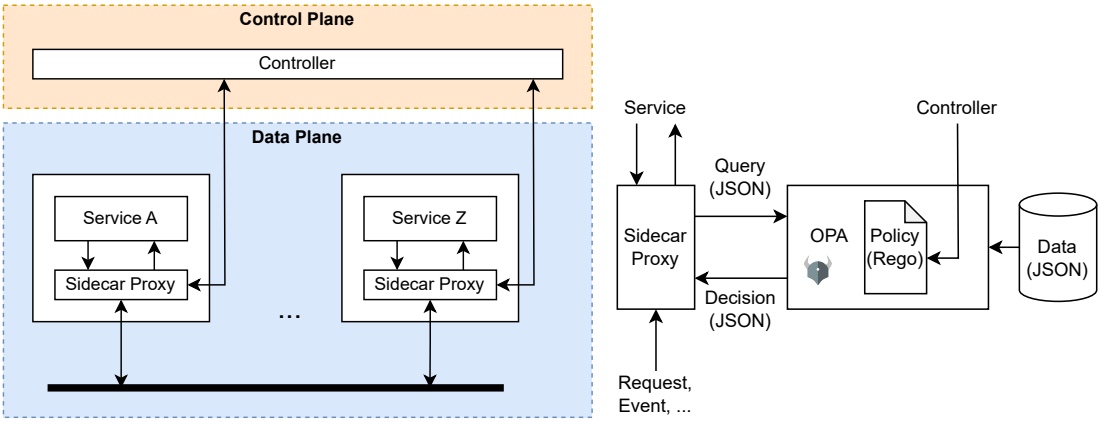
In microservice-oriented architectures, large monolithic applications are split into smaller, independent services, often implemented using virtual machines or *containers*. This approach offers numerous benefits, including scalability, resilience, and faster development cycles. However, it also introduces significant complexity, especially when managing inter-service communication and security. Hardcoding these functionalities into each service implementation introduces redundancy, complicates maintenance, and increases the risk of misconfigurations [6].

To address these challenges, *Service Mesh* (SM) technologies have emerged [4, 15]. A service mesh is a dedicated infrastructure layer designed to handle service-to-service communication. It provides a transparent *sidecar proxy* for each service, enabling features like load balancing, traffic management, security, and observability without requiring changes to the application code. By abstracting network complexity, service meshes decouple control functionalities from the core business logic of applications, enabling improved maintainability and governance, while ensuring reliable and secure communication.

While service meshes offer powerful capabilities, they demand effective governance mechanisms to maintain consistency, security, and compliance. To streamline governance and automate policy enforcement, *policy-based* approaches have gained traction [10, 11, 12, 13]. Policy-based governance leverages declarative domain-specific languages to define rules and constraints that govern the behavior of services. By separating policy from implementation, organizations can centralize policy management, ensuring consistency and reducing the risk of human error.

Among various domain-specific languages for policy authoring, Rego has emerged as a popular choice for service mesh environments [11]. Rego's expressive syntax and powerful evaluation engine enable the creation of sophisticated policies that can enforce a wide range of requirements, including security, reliability, and performance.

Rego's functional and declarative nature allows it to access a data store for policy evaluation, but it is limited to read-only operations. This restriction can hinder the specification of access policies that require maintaining and updating state. For instance, a policy enforcing a rate



(a) A generic service mesh representation. (b) OPA's decision flow.

Figure 1: Policy enforcement in distributed systems.

limit of 10 requests per hour from service B to service A necessitates tracking access counts and timestamps. Similarly, as in Bell-LaPadula model, a policy granting A access to service B based on B's lack of access to service C requires remembering the "B to C" access history.

These scenarios highlight the need for stateful policy enforcement, which is not directly supported by Rego's core capabilities. In fact, in these cases the state must be updated by the services, thus violating the separation between business logic and policy specification.

To address the limitations of Rego's stateless nature, in this paper we introduce the *Open policy agent Wrapper State Manager* (OWSM). OWSM maintains a separate state store to track policy-specific information, which can be accessed by Rego engine during policy evaluation. From Rego's JSON response, OWSM extracts state update instructions and forwards the final authorization decision to the OPA agent, sidecar to the real service. By leveraging OWSM, we can write stateful policies directly in Rego without modifying its syntax or evaluation engine. In this way, services do not need to update the state with policy-specific information, thus keeping business logic and policy specification well separated. Moreover, thanks to controlled access to avoid inconsistencies, the data store can be shared across multiple Rego engines, allowing for efficient and concurrent access to stateful policy-specific information.

Synopsis. In Section 2, we provide an overview of service meshes, OPA and Rego. To address its limitations, we introduce the *OPA Wrapper State Manager* (OWSM) in Section 3. In Section 4, we present the results of experiments conducted to evaluate the impact of OWSM. Finally, we conclude in Section 5, summarizing our findings and outlining directions for future work.

2 Background and related works

2.1 Service Meshes

A Service Mesh typically consists of two main components: a *controller* and a set of *sidecar proxies* (Figure 1a). The sidecar proxies form the *data plane* network, which directly handles the flow of requests between microservices. Each proxy is responsible for intercepting all incoming and outgoing traffic to its service, enforcing the policies received from the controller on a separate *control plane*.

SMs can have multiple goals, here we focus on their role in *authorization* and *authentication*. *Authorization* determines which services or users are permitted to perform specific actions on resources, while *authentication* ensures that only legitimate services can communicate with each other. Compared to traditional orchestrator policies, SMs enable more fine-grained access control and identity management. A key advantage of SMs is their centralized policy enforcement, which simplifies the management of security policies across a distributed environment. Policies are defined centrally and propagated to sidecar proxies, which locally enforce the policies by intercepting and evaluating requests, thus ensuring decentralized decision-making. For each intercepted request, a proxy evaluates the authorization policy against the received data and metadata, such as source service identity, user attributes, request headers, and requested actions. Based on the policy evaluation, the sidecar proxy decides whether to allow or deny the request. This enforcement occurs transparently to the services, ensuring security without requiring modifications to application code.

2.2 Security Policies in Distributed Applications

The challenge of expressing security policies in distributed applications has been a major focus of research and development in recent years. Existing solutions can be categorized into two approaches: *Policy via Configuration Files* and *Policy-as-Code*.

Configuration files are collections of key-value pairs used for defining system configurations and behaviors. This approach is widely adopted in cloud computing, including SMs. Notable examples include Istio [15] and Linkerd [7], which leverage configuration files to encode policies in a structured and reproducible manner. However, these files inherently lack support for conditional logic, complex data structures, and arithmetic operations. This limitation in expressiveness hinders their suitability for defining intricate or dynamic policy requirements.

On the other hand, the Policy-as-Code paradigm [12] advocates for expressing security policies through specialized programming languages. This approach, in contrast to traditional configuration files, enables complex constructs like arithmetic operations and conditional control flows, offering enhanced expressiveness and flexibility. A pioneering example of this paradigm is XACML [8], which represents access control policies using XML and XSLT files. Notably, the XACML specification does not encompass the design or implementation of authorization agents (there called *Policy Decision Points*). More recent proposals include OpenFGA [10], a fine-grained authorization engine drawing inspiration from Zanzibar [13] (Google's authorization system), and Cedar [2], a programming language for access control developed by AWS Labs, whose semantics is rigorously formalized and verified in Lean [3].

One of the most widespread language of this category is Rego [11], part of the Open Policy Agent (OPA) project. Rego manipulates semistructured data (in JSON format): when evaluating a request, Rego produces an object encapsulating the results of the policy evaluation, enabling detailed decision-making. Due to this flexibility, Rego has been used in various domains, ranging from cloud compliance automation to authoritative nameserver architecture [5, 9, 14]. Moreover, policies written in XACML can be translated to Rego, and vice versa. The widespread adoption and growing interest in Rego motivated us to focus on extending OPA's functionality, the official authorization engine for Rego, to address its current limitations.

A Rego policy is a collection of *rules* (see Figure 2). Each rule comprises a *head*, which defines the decision or value to be computed, and a *body*, which consists of a set of conditions or queries that must evaluate to true for the rule to be applicable. To make decisions, OPA can access not only the information provided in the request but also additional *data*, seamlessly integrated within the Rego language. This external data enriches the decision-making process.

```
package app.rbac
import rego.v1

# By default, deny requests.
default allow := false

# Allow admins to do anything.
allow if user_is_admin

# Allow the action if the user is granted permission to perform the action.
allow if {
        # Find grants for the user.
        some grant in user_is_granted
        # Check if the grant permits the action.
        input.action == grant.action
        input.type == grant.type
}

# user_is_admin is true if "admin" is among the user's roles as per data.
    user_roles
user_is_admin if "admin" in data.user_roles[input.user]

# user_is_granted is a set of grants for the user identified in the request.
# The `grant` will be contained if the set `user_is_granted` for every...
user_is_granted contains grant if {
        # `role` assigned an element of the user_roles for this user...
        some role in data.user_roles[input.user]
        # `grant` assigned a single grant from the grants list for 'role'...
        some grant in data.role_grants[role]
}
```

Listing 1: Example of Rego policy for RBAC.

```
{
        "user_roles": {
                "alice": ["admin"],
                "bob": ["employee", "billing"],
                "eve": ["customer"]
        },
        "role_grants": {
                "customer": [{"action": "read", "type": "dog"}, [...]],
                "employee": [{"action": "update", "type": "dog"}, [...]],
                "billing": [{"action": "read", "type": "finance"}, [...]]
        }
}
```

Listing 2: Data for the RBAC example.

```
{"user": "alice", "action": "read", "object": "id123", "type": "dog"}
```

Listing 3: Input for the RBAC example.

```
{"allow": true, "user_is_admin": true, "user_is_granted": []}
```

Listing 4: Output for the RBAC example for the provided input.

Figure 2: Example of a Rego policy for RBAC (see Section 4). From top to bottom: the policy, the data, an input for the policy evaluation and the corresponding output.

As Figure 1b shows, OPA performs the following steps to evaluate a policy: 1. OPA accepts JSON-formatted inputs representing the request; 2. OPA interprets Rego rules to compute a decision based on the request and data; 3. OPA returns the evaluation outcome to the requester as a JSON response, which contains the outputs of all the evaluated rules.

Despite the versatility of Rego, certain use cases remain challenging due to its stateless nature. Specifically, Rego lacks the ability to manage or persist *data* across requests, limiting its applicability in scenarios that require stateful paradigms. In fact, in these situations the update of the *data* object is on the programmer of the services, thus violating the separation principle between policy specification and business logic.

3 OWSM: OPA Wrapper State Manager

In this section we present *OPA Wrapper State Manager* (OWSM), whose aim is to tackle OPA's limitations by extending it with state management capabilities. By using OWSM, developers are freed from the burden of managing state directly within their application's business logic in scenarios where persistent state is crucial.

An example of an inherently stateful policy is the one regulating access to a service API where each call costs a token; each user is given a certain number of tokens at the beginning of the month. The policy has to keep track of token expenditure, and to reset token counters once a month. This can not be expressed in traditional policy engines without the introduction of additional elements that modify and maintain the number of available tokens.

Another example comes from security concerns, such as those investigated in [1, 18]. Let us consider three services A, B and C which run at different levels of clearance, e.g., A is at higher level than C; according to the Bell-LaPadula security model, we want to prevent data leakage from C towards A. To ensure this property, we want to forbid B's requests to communicate with C if B has previously communicated with A. Also in this case, we need to maintain the status of communication between services and this can be accomplished by traditional policy engines only with the introduction of additional stateful elements in the services themselves.

Our solution relies on wrapping the OPA engine with a custom API that interacts with a datastore in order to maintain and modify a *state* at runtime. Comparing with Figure 1a, a sidecar proxy will interact with an OWSM instance, permitting or not a request access. There can be multiple OWSM instances, one for each sidecar proxy in the service mesh.

3.1 Requirement definitions

A core requirement for our system is the inclusion of primitives for reading from and writing to a *state* that persists across multiple policy queries. These primitives enable a policy decision engine to dynamically update the data upon which decisions are made.

The state must be accessible by multiple instances of the decision engine. Consider a service mesh where decision engines are deployed as sidecar proxies alongside multiple replicas of a web server offering an API. If we aim to protect this API with the aforementioned "limited-token" middleware, the state must be maintained consistently across all replicas. This centralized state management is crucial to ensure that the policy "a user can access the API up to N times, where N is the number of tokens available to the user" is enforced uniformly across all replicas.

To achieve this, our system should consist of two components: (1) a policy decision engine that can interact with (2) a datastore. We choose OPA as the underlying policy engine and Rego as the policy language. This choice is based on the fact that its output is a JSON object, rendering it flexible and aiding the integration with our system.

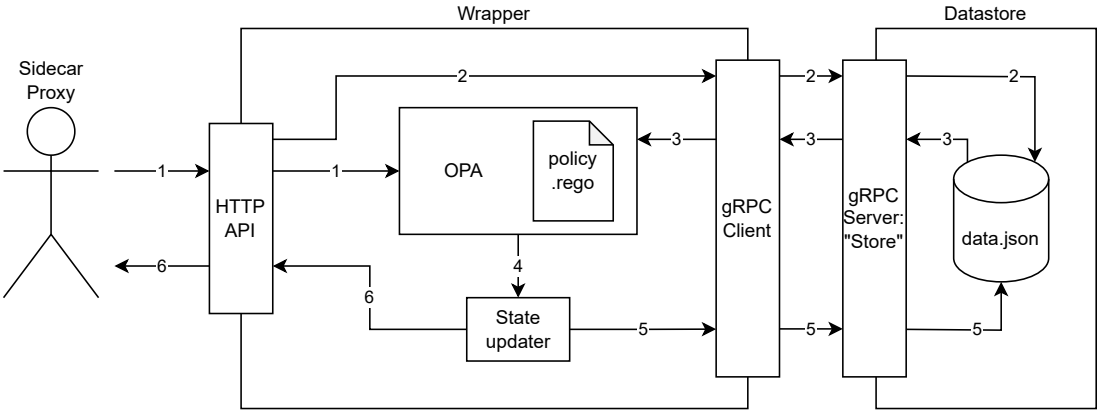


Figure 3: Diagram of OWSM implementation. The data flow is as follows: 1. the system receives a query from a service and adds it to OPA's evaluation context; 2. the wrapper asks the datastore for the current data, locking the datastore; 3. the datastore returns the data to OPA; 4. OPA evaluates the policy and returns the result, together with the data to be updated in the datastore; 5. the updated data is sent to the datastore, which is unlocked; 6. the result is returned to the user, stripped of the data sent to the datastore.

3.2 Design and Implementation

To avoid directly modifying the decision engine, we wrap it with another API that receives decision queries, gets the updated state from the datastore, and pass both of them to OPA. Figure 3 depicts our solution implementation.

The datastore must implement some locking mechanism to ensure consistency in presence of concurrent requests. The minimal functional API for the store must thus contain four endpoints: two to respectively get and set the state, and two to interact with the locking subsystem.

The modification of the state is achieved by reserving a special policy name, i.e., *state*; this policy can be defined by Rego rules (as any other policy), yielding a standard JSON dictionary. This output is interpreted by the wrapper, which extracts the keys that need modification in the datastore, removing them from the JSON. The datastore is updated accordingly, and then unlocked, before returning the decision result to the requesting service.

From an implementation point of view, we decided to use the Go programming language to implement both the wrapper and the datastore. This choice has been made due to the fact that OPA is written in Go and offers a native Go library to interact with its internals.

The API of the datastore is offered via gRPC which is a universal, open source, high performance Remote Procedure Call framework. More specifically, the datastore exposes a gRPC service, called `Store`, with four procedures: `Get`, to retrieve the state; `Put`, to update a key with a new value; `Lock`, to guarantee mutually exclusive access to concurrent clients; and `Unlock`, to release the lock. The wrapper communicates with the datastore through gRPC and, finally, exposes to the final user an HTTP endpoint to allow for submission of decision queries, as does OPA when used as an HTTP API. We use */query* as the endpoint name.

To validate our system, we implemented the "token counter" and "three microservices" use cases described above, and tested them using OWSM yielding a null error rate, thus proving the higher expressiveness of OWSM compared to OPA. See Figure 4 for the corresponding code.

```
package tokencounter
import rego.v1

default allow := false

allow if {
        input.user == "username"
        data.counter > 0
}

state["counter"] := data.counter - 1
      if allow
```

```
package threemicroservices
import rego.v1

# B can talk to C until A talks to B
default allow := false

allow if {
        input.source == "a"
        input.dest == "b"
}

allow if {
        input.source == "b"
        input.dest == "c"
        data.a_to_b == false
}

state["a_to_b"] if {
        input.source == "a"
        input.dest == "b"
}
```

Figure 4: The two implemented example policies for OWSM. The output for the *state* rule is intercepted by OWSM and used to update the datastore. On the left the "token counter" example, on the right the three microservices data leakage example.

4 Experimental Evaluation

In this section, we perform an experimental evaluation of OWSM to asses the efficiency and overhead introduced in comparison to pure OPA, i.e., without any stateful information.

4.1 Use cases

We consider use cases that can be expressed in both pure OPA and OWSM. These use cases are taken from the Access Control section of the Rego Playground [17].

Use case 1 considers an RBAC model for the Pet Store API [16], which allows users to view, adopt, and update pets. The policy governs which users can perform actions on specific resources, following a classic Role-based Access Control (RBAC) model. Users are assigned roles, which are granted permissions to act on certain resources.

Use case 2 follows an Attribute-based Access Control (ABAC) model for the same API, where users, resources, and actions are associated with attributes. Access decisions are made based on these attributes.

Use case 3 aims to mitigate the "Role Explosion" problem in RBAC through hierarchical roles. Role Explosion occurs when the number of roles in a RBAC grows exponentially as the number of users and permissions increases, leading to a complex and unmanageable set of roles. The example demonstrates how to implement a simple hierarchical access control policy using a graph of related roles. Hierarchical roles help address this issue by allowing roles to inherit permissions from other roles, reducing redundancy and simplifying role management. Specifically, users submit requests with one or more roles, and the policy checks if the user has the required permission by traversing the role hierarchy.

4.2 Research Questions and Methodology

RQ1 [**Time performance**] What are the characteristics of the overhead introduced by OWSM when handling sequences of requests?

Understanding the overhead introduced by OWSM is important to evaluate its impact on maintaining a good time performance during request handling. This overhead includes both the computational and latency costs of managing state modification and wrapping pure OPA.

RQ2 [**Concurrent scalability**] What is the behaviour of OWSM as the number of concurrent requests increases?

Understanding the behaviour of OWSM under increasing concurrency (i.e., number of concurrent requests to the datastore) is essential for evaluating its concurrent scalability. This requires analysing response times as the number of simultaneous requests grows.

To evaluate response timing, we rely on "Apache Benchmark" (`ab`) version 2.4.62, a standard tool for benchmarking web servers. We patched the source code of `ab` to support outputting timings in microseconds, instead of rounding them to the nearest millisecond. Our patch modifies the `ap_round_ms` macro to remove the rounding operation and return the raw runtime time value (already in microseconds) and does not modify the functionality of `ab`.

For comparing OPA and OWSM, we run experiments on both for each use case. At the beginning of each experiment, we perform a warmup of OPA and OWSM by running 10 non-concurrent requests. This avoids spurious high times from the first system query.

For RQ1 we run batches of 100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600, 51200 and 102400 requests to both OPA and OWSM, and we calculate the mean and the interquartile range (IQR) of the response times for each batch.

For RQ2 we run 50000 total requests with increasing levels of concurrency (100, 650, 1200, 1750, 2300, 2850, 3400, 3950, 4500, 5050, 5600, 6150, 6700, 7250, 7800, 8350, 8900, 9450 and 10000 parallel requests). We then calculate the mean and IQR of the response times for each concurrency level.

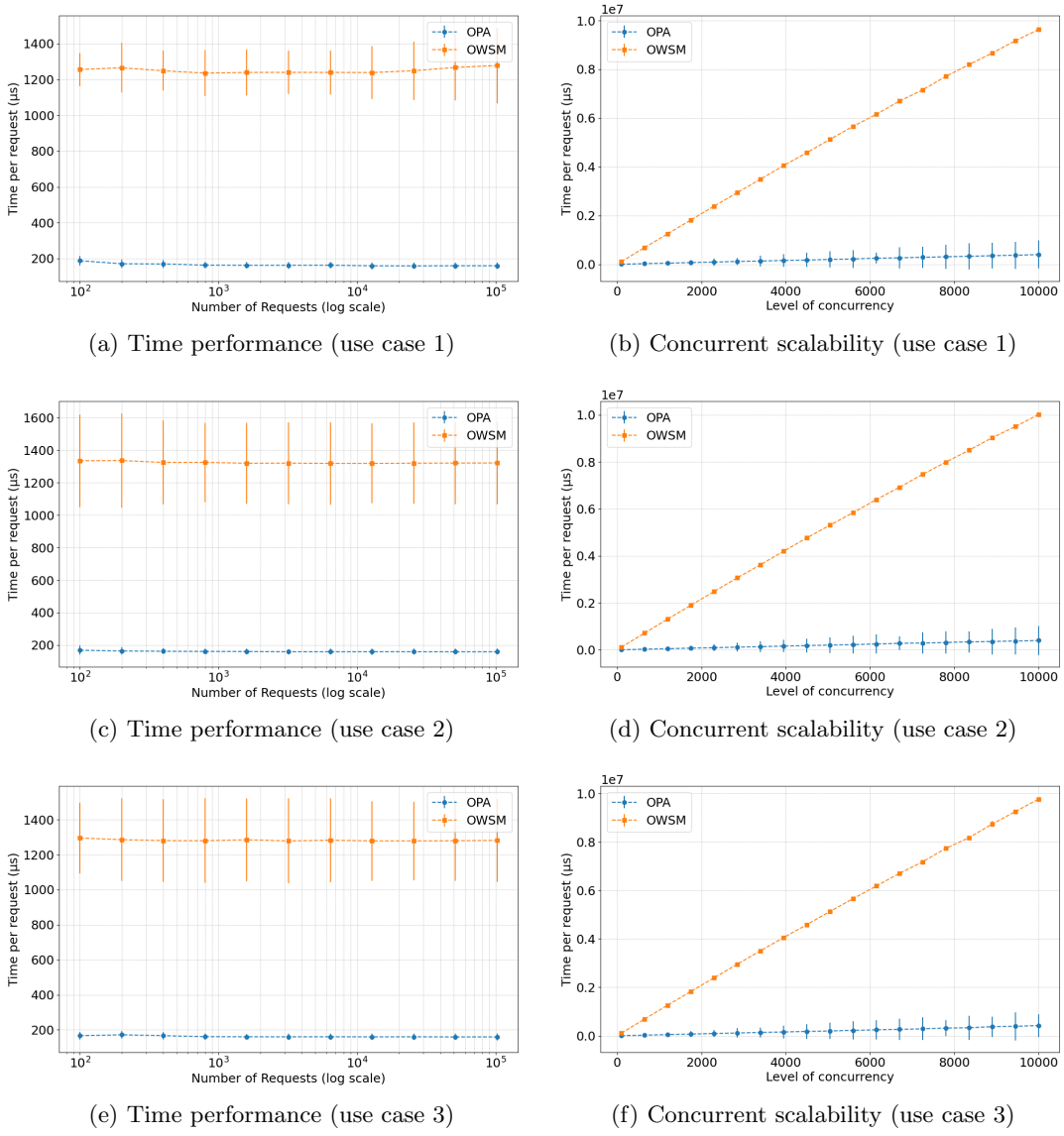
We use a server with Debian GNU/Linux 12 at kernel version 6.10-28 with a Intel(R) Core(TM) i9-10900 CPU @ 2.80GHz (20 threads) and 128 GB of RAM.

4.3 Threats to validity

To account for the inherent randomness in the measurements, we performed multiple requests in each configuration and assessed the statistical significance of the comparison between the IQRs of OPA and OWSM by using the Mann–Whitney U test, a non-parametric test used to compare differences between two independent samples. This test provides a measure of whether one group tends to have larger values than the other. Specifically, we calculated the U statistic and its corresponding p-value to determine if the observed differences between the systems were statistically significant at a significance level of $\alpha = 0.05$.

Our assessment of the observed trends is supported by fitting a linear regression model to the data and analysing the coefficient of determination (R^2).

We mitigate the threats to external validity (i.e. generalization) by considering three representative and diverse use cases directly drawn from the official OPA playground.



(a) Time performance (use case 1)

(b) Concurrent scalability (use case 1)

(c) Time performance (use case 2)

(d) Concurrent scalability (use case 2)

(e) Time performance (use case 3)

(f) Concurrent scalability (use case 3)

Figure 5: Comparison between OPA and OWSM.

4.4 Results

Figures 5a, 5c and 5e show the overhead introduced by OWSM over OPA in the three stateless use cases under consideration. Since OWSM consists of two separate parts (i.e. the wrapper and the datastore), which communicate over an API, it can introduce some variability in response times for individual queries. This is reflected in the significantly higher IQR shown by OWSM in the figures (p-value < 0.05).

However, the overhead remains nearly constant ($|\text{slope}| < 0.001$, p-value > 0.05) across all measured points. This is a desirable outcome, as it indicates that the overhead introduced by OWSM does not increase as the sequence of consecutive requests progresses. A constant

overhead means that OWSM can handle multiple consecutive requests without accumulating additional delays, ensuring consistent performance over time. This is important for maintaining predictability and reliability in real-world use cases, where repeated requests are common.

Answer to RQ1: OWSM introduces an overhead of ∼1 millisecond, which remains constant thorough the whole sequence of requests.

Figures 5b, 5d and 5f show that the overhead for concurrent requests introduced by OWSM follows a linear trend ($R^2 > 0.999$, p-value < 0.05). This is due to the fact that the implemented locking mechanism in the datastore allows only one request to be processed at a time, even when the store is not modified. This result is consistent with the previous experiment, as for 10000 concurrent requests the elapsed time per request is ∼10000 milliseconds.

Remarkably, OWSM demonstrates much more predictable response times in comparison to OPA, with a significantly lower IQR. In fact, the mean response time for OWSM is ∼10× lower than for OPA, highlighting its improved consistency in handling concurrent requests.

Answer to RQ2: At increasing concurrency level, the response time for OWSM increases linearly, with higher temporal stability.

The results from our experiments characterise the overhead introduced by OWSM over OPA, which is expected due to the added functionality. We believe that the temporal stability observed at increasing concurrency levels and the constant overhead during the handling of a sequence of requests demonstrate that OWSM is a promising solution to address the lack of state management primitives in OPA. However, the simple locking mechanism implemented in our prototype could be further improved. Future improvements could include more advanced mechanisms that would allow for some higher degree of concurrency while maintaining the observed temporal stability.

5 Conclusions

In this paper we introduced the OPA Wrapper State Manager (OWSM), a novel solution designed to extend the capabilities of the OPA authorization engine by incorporating state management for Rego policies. To validate the practical applicability of OWSM, we have successfully applied it to various scenarios that demand stateful access control. Our solution allows to maintain the definition of (stateful) access policies just as Rego rules, without the need of modifying the business logic of services.

To assess the performance impact of OWSM, we conducted empirical evaluations comparing it to pure OPA. Our findings demonstrate that OWSM maintains temporal stability even under increasing concurrency levels, while the introduced overhead remains relatively low, making it a viable solution for service mesh environments.

In our future work, we plan to generalize our findings to a broader range of use cases, including real-world scenarios derived from software repositories. Additionally, we aim to investigate more advanced locking mechanisms to enhance efficiency and enable higher levels of concurrency without compromising temporal stability. Moreover, our intention is to find a solution for verify some properties at static time for policies represented in REGO, avoid error and vulnerabilities in policy validation. Finally, an interesting direction of future research is how to integrate NLP in the definition and validation of stateful Rego security policies with respect to informal descriptions given in natural language.

Acknowledgments This work was partially supported by the PNRR M4C2 I1.3 "SEcurity and RIghts in the CyberSpace (SERICS)" PE0000014 PE7, CUP H73C22000890001, and the project SecCo-OC CUP D33C22001300002, both funded by Next-Generation EU.

References

[1] Valentina Casola, Vincenzo Riccio, Giuseppe Tricomi, Giovanni Merlino, Pietro Di Gianantonio, Bruno Crispo, Massimiliano Rak, and Antonio Puliafito. SecCO-OC: securing microservice-base apps. In *Proc. 10th Italian Conference on ICT for Smart Cities and Comunities*, 2024.

[2] Joseph W. Cutler, Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, Eleftherios Ioannidis, John Kastner, Anwar Mamat, et al. Cedar: A new language for expressive, fast, safe, and analyzable authorization. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):670–697, 2024.

[3] Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, John Kastner, Anwar Mamat, Matt McCutchen, Neha Rungta, et al. How we built Cedar: A verification-guided approach. In *Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 351–357, 2024.

[4] Mrittika Ganguli, Sunku Ranganath, Subhiksha Ravisundar, Abhirupa Layek, Dakshina Ilangovan, and Edwin Verplanke. Challenges and opportunities in performance benchmarking of service mesh for the edge. In *2021 IEEE International Conference on Edge Computing (EDGE)*, pages 78–85, 2021.

[5] James Larisch, Timothy Alberdingk Thijm, Suleman Ahmad, Peter Wu, Tom Arnfeld, and Marwan Fayed. Topaz: Declarative and verifiable authoritative DNS at CDN-scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 891–903, 2024.

[6] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 122–1225, 2019.

[7] Linkerd. The world's most advanced service mesh, 2024. Available at https://linkerd.io/.

[8] Markus Lorch, Seth Proctor, Rebekah Lepro, Dennis Kafura, and Sumit Shah. First experiences using xacml for access control in distributed systems. In *Proceedings of the 2003 ACM workshop on XML security*, pages 25–37, 2003.

[9] Reiya Oku, Kohei Shiomoto, and Yoshihiro Ohba. Decentralized identifier and access control based architecture for privacy-sensitive data distribution service. In *2022 IEEE 8th World Forum on Internet of Things (WF-IoT)*, pages 1–6, 2022.

[10] OpenFGA. Relationship-based access control made fast, scalable, and easy to use, 2024. Available at https://openfga.dev/.

[11] OpenPolicyAgent. Rego documentation, 2024. Available at https://www.openpolicyagent.org/docs/latest/policy-language/.

[12] Samodha Pallewatta and Muhammad Ali Babar. Towards secure management of edge-cloud IoT microservices using policy as code. In *European Conference on Software Architecture*, pages 270–287. Springer, 2024.

[13] Ruoming Pang, Ramon Caceres, Mike Burrows, Zhifeng Chen, Pratik Dave, Nathan Germer, Alexander Golynski, Kevin Graney, Nina Kang, Lea Kissner, Jeffrey L. Korn, Abhishek Parmar, Christina D. Richards, and Mengzhi Wang. Zanzibar: Google's consistent, global authorization system. In *2019 USENIX Annual Technical Conference (ATC '19)*, 2019.

[14] Alen Paul, Rishi Manoj, and Udhayakumar S. Amazon Web Services cloud compliance automation with Open Policy Agent. In *2024 International Conference on Expert Clouds and Applications (ICOECA)*, pages 313–317, 2024.

[15] Ozair Sheikh, Serjik Dikaleh, Dharmesh Mistry, Darren Pape, and Chris Felix. Modernize digital applications with microservices management using the Istio service mesh. In *CASCON '18: Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, page 359–360. IBM, 2018.

[16] Swagger. Swagger Petstore - OpenAPI 3.0, 2024. Available at https://petstore3.swagger.io/.

[17] Sytra. The Rego playground, 2024. Available at https://play.openpolicyagent.org/.

[18] Luca Verderame, Luca Caviglione, Roberto Carbone, and Alessio Merlo. SecCo: Automated services to secure containers in the DevOps paradigm. In *Proc. 2023 International Conference on Research in Adaptive and Convergent Systems, RACS 2023*, pages 10:1–6. ACM, 2023.