August 13, 2021

Massimo Mecella, Hakim Hacid, Hye-Young Paik, and Moha Naouel
Program Committee Chairs
International Conference on Service Oriented Computing

Dear Program Committee Chairs,

Re: ICSOC Minor & Resubmit of paper #7

First and foremost, we would like to thank the reviewers for the valuable feedback they provided for preparing a minor revision of the manuscript. We feel that their comments enabled us to considerably improve the quality of our manuscript and appreciate the time and effort that they have taken to indicate potential issues.

Attached to this letter, you will find the improved version of our manuscript titled "ThunQ: A Distributed and Deep Authorization Middleware for Early and Lazy Policy Enforcement in Microservice Applications". We extended our original manuscript with two additional pages of material and references in response to the reviewers' comments. Furthermore, this letter provides a detailed response to the comments provided by the reviewers and highlights the changes made to address their concerns.

If you require any additional information, please do not hesitate to contact us.

Sincerely,

ThunQ author team

Detailed answers to individual review comments and questions

The authors of the paper would like to thank the reviewers for their critical and insightful assessment of our work. Below we have provided detailed answers to the individual comments provided by the reviewers. We hope that these responses address the concerns raised by the reviewers about the original manuscript and welcome any additional feedback that can improve the manuscript further. We also uploaded an additional color-annotated version of our manuscript to highlight the changes in response of the reviewers' comments.

Reviewer 1

Reviewer Point P 1.1 — Authors should compare their approach with most recent related ones, such as:

- Hannousse, A., & Yahiouche, S. (2021). Securing microservices and microservice architectures: A systematic mapping study. Computer Science Review, 41, 100415
- Li, X., Chen, Y., Lin, Z., Wang, X., & Chen, J. H. (2021). Automatic Policy Generation for Inter-Service Access Control of Microservices. In 30th USENIX Security Symposium (USENIX Security 21)
- ShuLin, Y., & JiePing, H. (2020, October). Research on Unified Authentication and Authorization in Microservice Architecture. In 2020 IEEE 20th International Conference on Communication Technology (ICCT) (pp. 1169-1173). IEEE
- Pereira-Vale, A., Fernandez, E. B., Monge, R., Astudillo, H., & Márquez, G. (2021). Security in microservice-based systems: A multivocal literature review. Computers & Security, 102200

Response: We agree that the manuscript would benefit from a more extensive literature review. We included the papers referenced by reviewer 1 as a separate subsection of Section 5 *Related Work*, called *Securing Microservices*. In that subsection, we discuss some of the many security techniques proposed by literature to tackle the challenges of microservice security. In addition to discussing the specific works on access control in microservice applications provided by reviewer 1, we also used the provided surveys to give a more comprehensive overview of security techniques in microservice applications.

Reviewer Point P 1.2 — The description of the proposed middleware is too brief (or examples too simple)

Response: We acknowledge that parts of the middleware's discussion lacked details about implementation. We have added extra information regarding the implementation strategy of some of the core middleware components in subsections *Gatekeeper* and *Query Modifier* of Section 3 *ThunQ Middleware*. Additionally, we have clarified the way thunks are forwarded in the subsection on *Thunks* of Section 3.

Reviewer Point P 1.3 — The proposed architecture is based on many strong assumptions.

Response: While ThunQ tries to leverage the properties of a generic microservice application, it is true that it makes some strong assumptions in its security model, mainly regarding the trust towards the platform administrators. Although ThunQ itself does not deal with this specific issue, we included a reference in subsection *Securing Microservices* of Section 5 *Related Work*, that deals with the issue of integrity protection and insider threat in microservice applications. It should also be noted that a system for access control such as ThunQ should be combined with other industry-standard approaches to offer sufficient protection against various kinds of security treats. By defining a strict security model for ThunQ we aim to be clear and precise of what ThunQ has to offer in terms of security.

Reviewer 2

Reviewer Point P 2.1 — I strongly suggest to the authors to update the literature review by looking also for more recent approaches in the field.

- Antonio Nehme, Vitor Jesus, Khaled Mahbub, Ali E. Abdallah: Fine-Grained Access Control for Microservices. FPS 2018: 285-300.
- Preuveneers Davy and Joosen Wouter: Towards multi-party policy-based access control in federations of cloud and edge microservices. In 2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) 2019
- Li Xing and Chen Yan and Lin Zhiqiang and Wang Xiao and Chen Jim Hao: Automatic Policy Generation for Inter-Service Access Control of Microservices. In 30th USENIX Security Symposium 2021

Response: This concern was also raised by reviewer 1. Therefore, we included the additional related work proposed by reviewer 2 in the subsection *Securing Microservices* of Section 5 *Related Work*.

Reviewer Point P 2.2 — The authors do not provide the code of their approach, prototype, or just experimental results. Despite it is not mandatory (and sometimes it is simply not possible), it would very helpful both for the reviewing activity and for understanding. Lastly, it would also increase the value of the paper itself.

Response: We have already opensourced the code on Github. However, as our code and prototypes contain personal references, we did not yet include it as a reference in the paper to maintain the double blind review. Should our work be accepted for the conference, we will include a reference to the code repositories in the camera ready version of the paper.

Reviewer 3

Reviewer Point P 3.1 — I would not mention security as a general term. The paper does already discard attacks like impersonation, or direct attacks against the databases. I would mention access control, as it is more precise with the goal of the article

Response: It is indeed true that the general use of the term *security* can be confusing to the reader. Therefore, we replaced generic terms such as *security policy*, *security architecture*, *security analysis* and others with the more specific terms *authorization policy*, *authorization architecture* and *access control analysis*. We also replaced the more general *access control* with *authorization* as a more specific term where it was applicable.

This comment was also one of the main reasons to discuss additional security techniques for microservice applications in the subsection *Securing Microservices* of Section 5 *Related work*, where we discuss different techniques for securing microservice applications put forward by literature.

Reviewer Point P 3.2 — A clarification should be added for the case, where the request needs data from two domains. Is the partial evaluation passed to every microservice, or do the attributes of one domain only travel towards the microservice of such a domain? In case of the former, I would like to know, how the overhead of processing such requests affects the architecture. In case the authors consider such overhead to be insignificant, I would suggest to explain the reason behind this

Response: To address these concerns, we added a clarification to the discussion of *Thunks* in Section 3 *ThunQ Middleware*, explaining that thunks are forwarded in their entirety by the microservices to other services further downstream. This was a deliberate design decision to promote loose coupling between the different microservices and the security policies. Splitting thunks according to the target domain would require that the calling microservice knows the policies that are applicable to both the target microservice and the microservices further down the request call chain.

We discuss the performance cost of this decision in the subsection *Discussion* of Section 4 *Evaluation*, we assumed that the overhead of forwarding the entire thunks is small in most application scenarios for the following reasons. First, thunks are reduced by the policy engine and are oftentimes smaller versions of the original application policies, which reduces the total size of the thunks. Second, generalizing our evaluation results, we can assume that the search queries are the dominant source of overhead in most target systems, outweighing the overhead caused by the inflated thunks.

Reviewer Point P 3.3 — I would suggest to explain the impact of ThunQ in terms of memory in the different microservices. As it appears across the whole architecture in form of different components, an explanation of how each of them affects each instance could be valuable.

Response: We agree that the issue of memory consumption has been left largely unaddressed by our previous submission. Therefore, we have improved the manuscript by adding additional information regarding the implementation and expected memory consumption of ThunQ's main components in subsections *Gatekeeper* and *Query Modifier* of Section 3 *ThunQ Middleware*.

In subsection *Gatekeeper*, we added implementation details of the *gateway filter*, which is implemented as a stateless component and should therefore have a small and constant impact on the memory consumption of the API gateway. The *policy engine* component of the *gatekeeper*, on the other hand, is an off-the-shelve instance of the Open Policy Agent (OPA). ThunQ uses OPA to partially evaluate policies and create a reduced version of the original authorization policies. The memory consumption of OPA is dependent on the specific policies evaluated and the data required for their evaluation. In our application case, we used OPA as a stateless sidecar, supplying it with all information necessary for policy evaluation. This configuration led to a limited memory overhead of maximum 10 MiB for all the policies used in our evaluation scenarios. However, for a stateful deployment, the memory usage could be considerably larger, as more information is kept in-memory. Therefore, we mention in subsection *Gatekeeper* of Section 3, that ThunQ can be deployed with OPA as a sidecar of the API gateway or as a standalone service, such that users can deploy OPA in a manner that best fits their use-case.

In subsection *Query Modifier*, we added a description of the implementation of the *query modifier* component. The query modifier is implemented as a stateless component and should therefore have a relatively small impact on the memory consumption of the microservices.

Reviewer Point P 3.4 — I think that the terms "high selective policies" and "high selectivity in policies" can be confusing for the reader. For that reason, I would remove high selective policies, and only use selectivity, as it already appears in the figure.

Response: We recognize that the use of *selective policies* and *selectivity* can be confusing to the reader. Therefore, we replaced the terms *selective* and *permissive* policies with *narrow* and *broad* respectively. This should avoid confusion for the reader. However, we opted to keep separate terminology for policies with low hand high values for selectivity as we felt that this made the discussion clearer.

ThunQ: A Distributed and Deep Authorization Middleware for Early and Lazy Policy Enforcement in Microservice Applications

Resubmission of paper #7 (Annotated Version)

Abstract. Online software services are often designed as multi-tenant, API-based, microservice architectures. However, sharing service instances and storing sensitive data in a shared data store causes significant security risks. Application-level access control plays a key role in mitigating this risk by preventing unauthorized access to the application and data. Moreover, a microservice architecture introduces new challenges for access control on online services, as both the application logic and data are highly distributed. First, unauthorized requests should be denied as soon as possible, preferably at the facade API. Second, sensitive data should stay in the context of its microservice during policy evaluation. Third, the set of policies enforced on a single application request should be consistent for the entire distributed control flow.

To solve these challenges, we present ThunQ, a distributed authorization middleware that enforces authorization policies both early at the facade API, as well as lazily by postponing authorization decisions to the appropriate data context. To achieve this, ThunQ leverages two techniques called partial evaluation and query rewriting, which support policy enforcement both at the facade API, as well as deep in the data tier.

We implemented and open-sourced ThunQ as a set of reusable components for the Spring Cloud and Data ecosystem. Experimental results in an application case study show that ThunQ can efficiently enforce authorization policies in microservice applications, with acceptable increases in latency as the number of tenants and access rules grow.

1 Introduction

Contemporary online services often provide a customer-facing API and adopt an internal architecture based on application-level multi-tenancy and microservices. Application-level multi-tenancy [10], as illustrated in Fig. 1, benefits from economies of scale by sharing resources between the tenants, such as the application and database. However, storing sensitive tenant data poses significant security risks. Application-level access control [34] is a key security technique that mitigates these risks by enforcing authorization policies at the application-level to block unauthorized access to resources. Moreover, multi-tenant applications require that both the application provider and tenants can specify these policies. In particular, the provider specifies the basic authorization policies for the platform, while the tenants can provide additional policies that further restrict

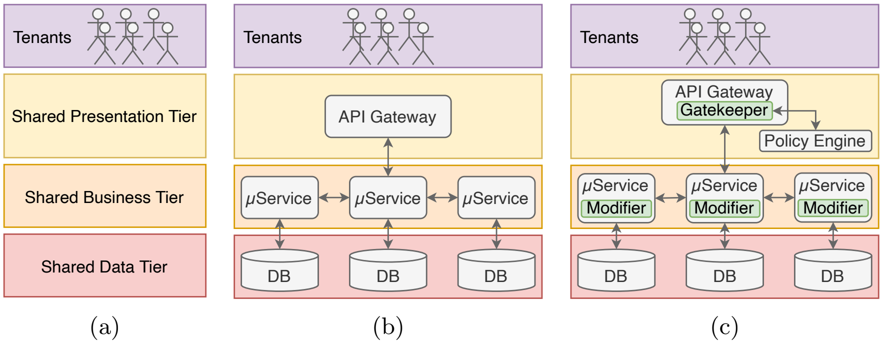


Fig. 1. Overview of application-level multi-tenancy (a) for both microservice applications (b) and applications with ThunQ (c). ThunQ's components are shown in green.

access by their end-users to comply with internal authorization policies. For example a tenant policy may state that: *"An insurance company employee can only view insurance documents of customers that are assigned to the employee."*

Supporting tenant specific policies requires an appropriate level of modularity, separation of concerns and adaptation of the related software artefacts [8]. While single-tenant applications can embed the authorization logic directly in the database query to enforce fine-grained access control, it is no longer feasible for multi-tenant applications with custom authorization policies per tenant. Custom policies require a more flexible approach where policies can be updated at run-time, as new tenants are continuously added to the application.

A frequently used architectural pattern to realize multi-tenant applications are *microservices* [23]. Microservice applications often adopt the *API gateway* [32] and the *database-per-service* [32] pattern as shown in Fig. 1b. The distribution of application logic and data in multi-tenant microservice applications introduces the following new challenges for access control in such applications:

1. Unauthorized requests should be denied *as soon as possible* (ASAP), such that unauthorized resource usage and control flows in the distributed microservice application are minimized.
2. Sensitive data should stay in the context of its microservice during policy evaluation, i.e. data from the data tier should not flow to the API gateway when evaluating authorization policies.
3. The set of policies enforced on a single application request should be consistent for the entire distributed control flow, as policies are no longer only enforced at the facade API but throughout the entire application.

Existing work on application-level access control [15,18,20,29,34] and API gateways [29,44,45] aims to enforce authorization policies ASAP, resulting in a permit or deny. However, these solutions require that sensitive data is brought outside of its microservice context. Other related work focuses on enforcing access control in application databases [3,16,24]. These solutions aim to restrict access by enforcing fine-grained authorization policies on the data records by either rewriting the original database query [3], defining authorization views [24] or by filtering database records after retrieving them from the database [16]. However, securing database access is only a part of the challenges to enforce a consistent set of authorization policies over a large number of microservices.

To address the challenges and shortcomings above, we present *ThunQ*, a distributed authorization middleware for multi-tenant microservice applications designed to efficiently and consistently enforce a set of authorization policies on distributed application services and data. ThunQ enforces authorization policies early in the distributed control flow, as well as deep down in the *data tier*. ThunQ achieves this by adding the *gatekeeper*, *policy engine* and *query modifier* components to the generic microservice architecture as shown in Fig. 1c. The gatekeeper and policy engine use partial policy evaluation [26] to create *thunks* that are piggybacked on the application request. The thunks are then used by the query modifier to enforce authorization policies deep in the data tier.

We implemented and open-sourced ThunQ as a set of reusable components for the Spring Cloud and Data ecosystem. Our evaluation shows that ThunQ performs notably better than state-of-practice postfiltering approaches. Moreover, ThunQ's overhead is largely independent of the number of application tenants and the complexity of the tenant specific policies.

The remainder of this text is structured as follows. Section 2 presents the motivational use case and provides the reader with background on access control and ThunQ's supporting technologies. Section 3 presents the architecture and the security model of the ThunQ middleware. Section 4 discusses the evaluation and results. Section 5 discusses related work and Section 6 concludes this work.

2 Motivational Use Case and Background

This section presents the motivation and background for ThunQ. We start with presenting *e-insurance*, an anonymized industrial case study of a multi-tenant insurance brokering platform with a microservice architecture and API-based online service offering. Next, we discuss background on access control models and ThunQ's enabling technologies.

The E-Insurance Case Study. In the financial industry, insurance companies or insurers do not always sell their insurance products directly to end customers. Instead, they employ intermediaries, called insurance brokers, to bring their products to the customer. Brokers negotiate insurance contracts with the customers and take care of the paperwork related to the contract. Furthermore, customers should have access to information regarding their insurance products, such as the current balance of their life insurance account. As shown in Fig. 2, e-insurance integrates insurers, brokers, and customers into a single platform that shares their insurance documents. E-insurance is responsible for storing the insurance contracts and their related documents, as well as offering advanced

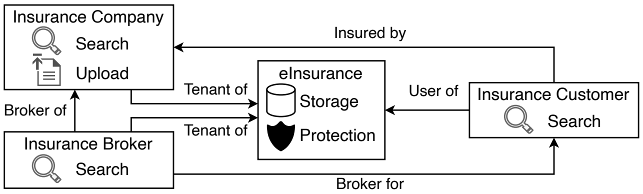


Fig. 2. Participants of the e-insurance application.

search operations on stored documents. However, as the contents of the insurance documents are sensitive, the results of the search operations should only include the information which the user is authorized to view.

Access Control Analysis. Ensuring the confidentiality of the insurance documents is the primary security goal of e-insurance. To achieve confidentiality, e-insurance must restrict access to only those users who are authorized to access a given document. Whether or not a user is authorized to access a document is determined by *authorization policies*. E-insurance defines two sets of policies: platform policies which are specified by e-insurance itself, and tenant policies, which are specified by the tenants to further restrict access by their end-users. Next, we provide a sample of possible policies.

P1. (platform) Brokers can only view documents assigned to them.
P2. (platform) Customers can only view documents that belong to them.
P3. (broker) Only senior employees can view documents worth over $100k.
P4. (insurer) Employees can only view the documents assigned to them.
P5. (insurer) Employees can only view documents during working hours.

Challenges. Given the discussion above, we can identify the following challenges for e-insurance. First, the application must guarantee the confidentiality of insurance documents by enforcing both platform and tenant policies. Second, e-insurance must offer the performance necessary to support numerous tenants and documents. Searching documents should be fast even as the number of tenants and documents increases. Finally, the set of policies applied to a single application request should be consistent for the entire distributed control flow.

Background. Access control models are models that determine which subjects, such as users and processes, are authorized to access a given object, such as files and other resources. The choice of access control model has a significant impact on the kind of authorization policies that can be expressed. Examples of access control models include Lattice Based Access Control [27] and Role Based Access Control [28]. We focus on Attribute-Based Access Control (ABAC) [12] in combination with Policy-Based Access Control (PBAC) [25]. ABAC models access rights by assigning attributes to the subjects and objects. ABAC makes authorization decisions dynamically, based on the assigned attributes and the environment, such as location and time. PBAC, on the other hand, makes decisions based on authorization policies. These policies are evaluated by a policy engine that uses an access control model, such as the attributes and context assigned by the ABAC model, to reach an authorization decision.

The separation of concerns between authorization policies and the mechanism to enforce them is a key principle in secure software engineering [8]. PBAC [25] decouples policy from mechanism by using policy engines to evaluate policies written in authorization policy languages. The *Open Policy Agent* (OPA) [41] is a policy engine that supports the *Rego* [40] policy language for writing policies. Rego policies use the attributes provided by the authorization request, as well as the access control model stored by OPA. OPA supports both full and *partial evaluation* [26] of authorization policies. Partial evaluation reduces a given policy by substituting the known variables in the policy and evaluating the involved expressions. The result of a partial evaluation is a reduced version of the original policy that only contains unknown variables. We further refer to the reduced version of the policy as the *residual policy*.

The OASIS eXtensible Access Control Markup Language (XACML) [18] is an industry standard for access control. XACML provides a specification for the XACML policy language and a reference architecture for authorization systems. XACML combines PBAC and ABAC, using XML documents to specify authorization policies. The XACML reference architecture contains the following components: (i) a Policy Enforcement Point (PEP), which intercepts incoming application requests, (ii) a Policy Administration Point (PAP), that manages the system's policies, (iii) a Policy Information Point (PIP), that stores the access control attributes, and (iv) a Policy Decision Point (PDP), which takes authorization decisions based on the context provided by the PAP and PIP.

3 ThunQ Middleware

This section presents ThunQ, a distributed authorization middleware for multi-tenant microservice applications. ThunQ is designed to efficiently enforce a consistent set of authorization policies on distributed application services and data. ThunQ combines *partial policy evaluation* [26] and *query rewriting* [2,3] to enforce authorization policies both *early* and *lazily*. Early enforcement denies unauthorized requests as soon as possible, while lazy enforcement pushes access decisions further down the distributed control flow. Next, we define ThunQ's security model, followed by a description of the architecture and its key elements.

Security Model. Fig. 1b depicts the system model for applications supported by ThunQ. ThunQ assumes that all application requests pass through an API gateway [32], which is a *facade* for the services in the *business tier*. Microservices in the business tier execute the actual business logic of the application and can call other microservices. Additionally, the services in the business tier rely on the databases in the *data tier* for persistence. ThunQ supports dedicated databases per service, as well as a single database that is shared between microservices. Given this system model, ThunQ makes the following trust assumptions.

A1 All services shown in Fig. 1b are trusted and operate correctly.

A2 Policies defined by the platform's security administrators are correct, meaning that they enforce the intended security policies.
A3 Tenant policies do not impact existing security properties of the system, i.e. policies are defined by the provider's security consultant after a requirements analysis of the tenant.
A4 Security administrators are trusted, i.e. there is no insider threat caused by the security staff.

The primary security goal of ThunQ is to restrict access to the distributed application logic and data by enforcing platform and tenant policies. First, ThunQ should deny unauthorized requests as soon as possible. Second, ThunQ should enable the confidentiality of application data by enforcing the authorization policies on individual data records deep in the data tier. ThunQ only achieves these goals when the following assumptions about the attacker hold.

A5 An attacker can only interact with the system through the APIs provided by the platform.
A6 An attacker cannot impersonate any other user.
A7 The attacker has no access to side-channels in the communication between the system and the attacker.

ThunQ's Overall Architecture. The authorization architecture of ThunQ is shown in Fig. 3. ThunQ adds the following components to realize its security goals. First, ThunQ adds the *gatekeeper* to the API gateway. The gatekeeper performs authorization checks and piggybacks the *thunks* on the application request. Second, ThunQ transparently adds a *query modifier* to the microservices. The modifier intercepts database queries from the application and rewrites them to enforce authorization policies. Next, we discuss the application request flow with distributed policy evaluation, followed by ThunQ's core architectural elements.

Distributed Policy Evaluation. Policy evaluation in ThunQ is distributed, early and lazy. Evaluation is distributed, as ThunQ evaluates policies at different points in the microservice application, early, as unauthorized requests are denied ASAP by partial evaluation, and lazy, as ThunQ postpones access decisions by piggybacking the residual policies to the appropriate data context. More specifically, policy evaluation in ThunQ starts at the API gateway where incoming application requests are intercepted by the gatekeeper (1). The gatekeeper then inspects the request and extracts any information regarding the subject. Next, the gatekeeper selects the policies applicable to the request and calls the *policy engine* with the subject information and the selected policies as arguments (2). The policy engine then partially evaluates the policies and returns the *residual policies* to the gatekeeper (3). The gatekeeper transforms the residual policies into a thunk and attaches the thunk to the application request. Alternatively, the policy engine returns a deny, in which case the gateway blocks the request.

Next, the API gateway forwards the request to the relevant microservice (4.1). The microservice then handles the request either by querying the database

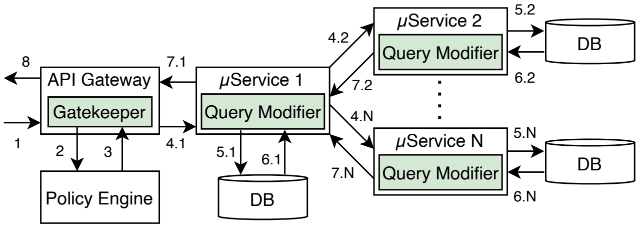
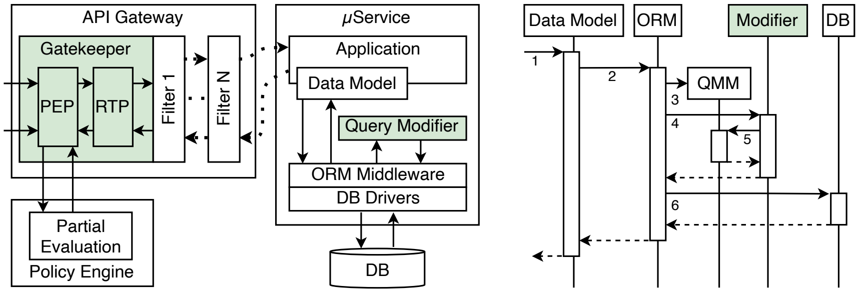


Fig. 3. Authorization architecture. ThunQ's components are shown in green.



(a) Components

(b) Query execution flow

Fig. 4. Detailed view of ThunQ's interactions with the application components.

(5.1 - 6.1) or by calling other microservices and piggybacking the thunk (4.x - 7.x). Each query made by the application gets intercepted by the query modifier, where the query gets rewritten to enforce the authorization policies before being passed to the database (5.1). The result of the rewritten query is then sent back to the application (6.1). After the data is retrieved, the application can perform other operations, eventually finishing the request and replying to the caller (7.1). Eventually, the API gateway receives the response and forwards it to the client (8). Note that the same rewriting procedure (5.x - 6.x) is applied when the service calls other microservices to handle the request.

We next discuss the core architectural elements of the ThunQ middleware. The ThunQ middleware consists of two main components the gatekeeper and the query modifier. These components and a policy engine are added transparently to the microservice application as shown in Fig. 4.

Gatekeeper. The gatekeeper enforces the authorization policies on the requests both early and lazily. As depicted in Fig. 4a, the gatekeeper is attached to the API gateway as a filter component that intercepts all incoming application requests. The gatekeeper can be further broken down into the *Policy Enforcement Point* or PEP, and the *Request Transformation Point* or RTP. The PEP is a modified version of a XACML PEP [18] and is responsible for sending requests for partial policy evaluation to the policy engine. The policy engine responds

```
1  allow {                            allow {
2    user.tenant=="insurer"             doc.tenant_id==67
3    doc.tenant_id==user.tenant_id      doc.employee_id==42
4    user.role=="account_manager"     }
5    doc.employee_id==user.id
6  }
```

Fig. 5. Example policy (left) and the residual policy after partial evaluation (right).

with either a set of residual policies or a deny. In the case of a deny, the PEP blocks the application request, denying the request early. Alternatively, the policy engine responds with a residual policy, in which case the PEP sends the residual policies to the RTP, which transforms the residual policies into Boolean expressions and adds the expressions to the thunk. The RTP is a new component in the XACML dataflow that is responsible for augmenting application requests, in particular by attaching a thunk for lazy enforcement.

Fig. 5 shows an example of partial policy evaluation at the gateway. The policy consists of rules which are defined by the provider at lines 2 and 3, as well as by the tenant at lines 4 and 5. Note that all subject attributes are available at the gateway such that lines 2 and 4 can be evaluated and, if necessary, denied early. This while lines 3 and 5 must be evaluated lazily in the data tier, as the attributes of *doc* are not accessible from the current evaluation context.

We realized ThunQ's gatekeeper as a *gateway filter* instance for *Spring Cloud Gateway* [44]. The gateway filter is implemented as a *stateless* instance to minimize ThunQ's memory footprint. However, the concept of the gatekeeper is more general and is not limited to this specific software implementation. The policy engine is provided by *Open Policy Agent* (OPA) [41], as it supports partial policy evaluation. OPA can be deployed as either a standalone service or a sidecar of the API gateway, depending on its memory consumption. For e-insurance we deployed OPA as a stateless sidecar, as memory use was limited to 10 MiB.

Thunks. A thunk is the key data structure that enables lazy and consistent policy evaluation in a distributed control flow. Thunks are created by the RTP which transforms the residual policies forwarded by the PEP into Boolean expressions. These expressions are added to a thunk by the RTP and piggybacked on the request. By piggybacking the thunks, the residual policies are able to travel together with distributed control flow, where they can be used by other ThunQ components to enforce fine-grained authorization policies deep in the data tier. As shown in Fig. 6, a thunk is a collection of *URL path selectors* mapped to a Boolean expression. The selectors are used by the query modifier to determine which residual policies are relevant for the intercepted database query. To ensure loose coupling, thunks are forwarded in their entirety between microservices. Note that each application request is processed with a consistent set of policies, as the same thunk is re-used for the entire the distributed control flow.

Query Modifier. The query modifier rewrites database queries such that the queries enforce authorization policies on individual data records. Note that the

```
{
  "/accountStates/*":"doc.tenant_id=67 && doc.employee_id=42",
  "/hospitalBills/*": <BoolExpr#2>,
  "/*": <BoolExpr#3>
}
```

Fig. 6. Example of a thunk encoding the partial policy of Fig. 5 and others.

```
SELECT *                 SELECT *
FROM account_states      FROM account_states
                         WHERE tenant_id=67 AND employee_id=42
                           AND <BoolExpr#3>
```

Fig. 7. Example of query rewriting by the query modifier. The original query on the left is rewritten using the thunk in Fig. 6 with */accountStates/all* as request path.

query modifier only augments search queries since these operate on large result sets. As shown in Fig. 4a, the query modifier is attached to the application as a plugin for the *Object Relational Mapper(ORM) middleware.* ORMs often provide hooks that enable third-party extensions to modify database queries through the *query meta-model* (QMM).

To rewrite queries, the query modifier must first determine the relevant residual policies to enforce. These policies are encoded as Boolean expressions in the thunks that are piggybacked on the application requests. The relevant Boolean expressions are selected by matching the URL path selectors of the thunk against the application request path. The matching expressions are then joined using a conjunction to create a Boolean expression that encodes all the matched residual policies at once. This expression is then woven into the meta-model of the database query by adding the expression to the *predicate* of the query's model. The modified query then gets further processed by the ORM middleware before it is sent to the database. The result of the query then is sent back to the ORM without passing through the modifier. An example of the effect of query rewriting on a SQL query is illustrated in Fig. 7.

Fig. 4b shows the flow of a database query in detail. First, the application invokes a search method on the data model (1). Next, the data model contacts the ORM middleware (2) which creates a query meta-model that corresponds to the method call (3). This meta-model is an internal representation of the query that the ORM will map later to a database specific query. Next, the ORM passes the meta-model to the query modifier (4), which rewrites the query as described earlier using the meta-model (5). After calling the modifier, the ORM instantiates the actual database query using the modified meta-model (6) and returns the result back to the data model. ThunQ's query modifier was realized as a component for the *Spring Data* [43] ORM middleware. The query modifier utilizes the Querydsl [35] query meta-model to rewrite database queries. Furthermore, the query modifier is implemented as a stateless component to minimize ThunQ's memory footprint.

4 Evaluation

This section discusses the evaluation of the ThunQ middleware with a key focus on the performance overhead of the middleware solution. We compare ThunQ against two alternative approaches for fine-grained authorization in the data tier, namely *postfiltering* [16] and *hand-crafted queries*. Postfiltering enforces authorization policies on data queries by checking each record in the result set against a policy engine. Hand-crafted queries, on the other hand, encode the authorization policies directly in the application queries. Although the last approach is impractical for multi-tenant applications, it represents the best-case scenario for query-based approaches to enforce fine-grained authorization, as it doesn't have the overhead of ThunQ's middleware components. The evaluation aims to answer the following questions related to multi-tenancy and performance.

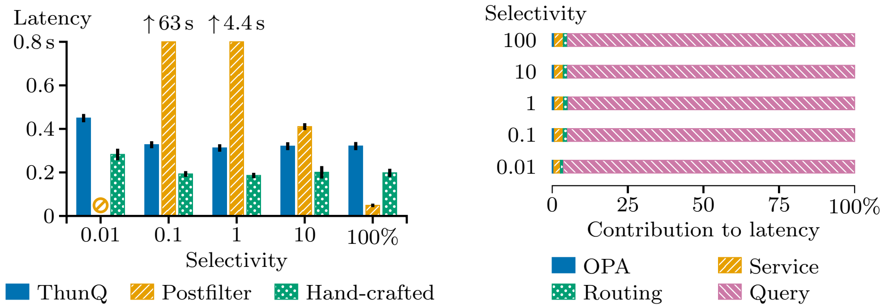
Q1 What is the impact of the properties of the enforced policies on the latency? As tenants specify policies that further restrict access by their end-users, it decreases the number of records included in the results. Also, adding policies can increase the number of attributes required for evaluation.
Q2 What is the impact on end-to-end latency when the number of tenants grows? As microservice applications are very sensitive to increases in latency, the overhead of ThunQ should not put limitations on the number of tenants.

Evaluation Setup. All experiments were performed on a proof-of-concept application (PoC) that is based on the e-insurance case study discussed in Section 2. The PoC was deployed in an AKS Kubernetes cluster in the Microsoft Azure public cloud. The Kubernetes control plane was hosted on a single Standard_B2s VM with 2 CPUs and 4GiB of memory, while the PoC runs inside a node pool consisting of 3 Standard_D4as_v4 VMs with 4 CPUs and 16GiB of memory. To simulate application users, we used the Locust [6] load generation tool.

The PoC consists of the following services: an *API gateway*, an *account-state service*, a *datastore*, and an *IAM* system. The API gateway is an instance of Spring Cloud Gateway [44] with an additional *gatekeeper* filter as discussed in Section 3. The account-state service handles statements of account balances generated by life insurances. The service is realized a Spring Boot [42] application augmented with the *query modifier* from Section 3. Furthermore, the datastore is an instance of Azure SQL and the IAM system is provided by Keycloak [39].

Q1. We first investigate the impact of two policy properties called *policy selectivity* and *attribute count*. Policy selectivity is the ratio between the number of data records still included after applying the policy to the result set and the size of the original result set. Policies with low values for selectivity are called *narrow*, as only a small portion of the original result set is included. Policies with high selectivity values are called *broad* as more records remain included. The attribute count of a policy, on the other hand, defines how many attributes are required by a policy for lazy evaluation.

We configured the experiments as follows. Clients send requests through the API gateway to fetch data from the account-state service, which has a database



(a) Average end-to-end latency

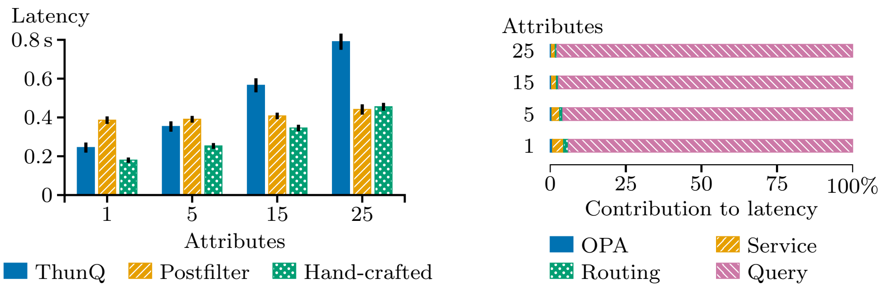
(b) Breakdown of ThunQ latency

Fig. 8. Latency in function of policy selectivity.

with 1 million records. Application requests are paginated and retrieve only the first 50 accessible records that satisfy the authorization policies. The policies in both scenarios were synthetically generated to show the impact of the different policy properties. The policies for the experiments with varying policy selectivity only have a single attribute, while the experiments with varying attribute count have policies with a selectivity of 10%.

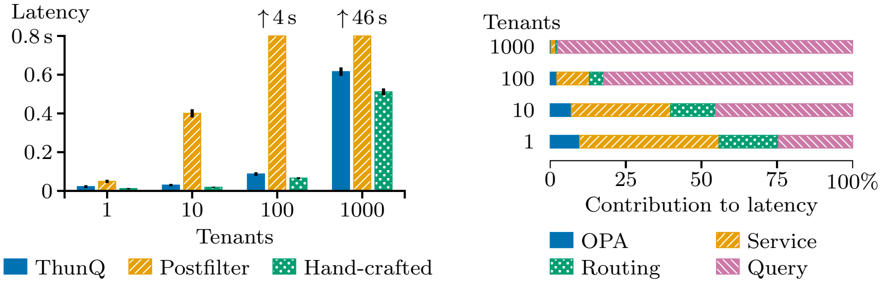
Impact of Policy Selectivity. Fig. 8a shows the impact of policy selectivity on the end-to-end latency. For ThunQ and hand-crafted queries, latencies are largely unaffected by policy selectivity, with only a minor increase for very narrow policies. In addition, the breakdown of the ThunQ's request latency shown in Fig. 8b, indicates that ThunQ's latency is dominated by the database query. The results for postfiltering show low latencies for policies with selectivity between 10 and 100%. This is a consequence of paged requests, as filling a page requires that only a limited number of records have to be checked against the policy engine. In contrast, narrow policies have high latencies. The decrease in selectivity means that more database records need to be checked by the policy engine before a single page can be filled, in turn increasing the overhead of the postfilter and the overall latency. A final observation concerns the results for policies with a selectivity of 100%. In this case, postfiltering outperforms both ThunQ and hand-crafted queries. This is caused by the way Spring Data handles request paging for ThunQ and hand-crafted queries.

Impact of Attribute Count. Fig. 9 shows the relation between the number of attributes used in the lazy evaluation of a policy and the end-to-end request latency for policies with a 10% selectivity. All three fine-grained authorization methods show a linear increase in latency for higher attribute counts. Although postfiltering initially performs worse than the other techniques, its slope is less steep compared to ThunQ or hand-crafted queries. Consequently it matches or outperforms the other solutions for higher attribute counts. The steeper slope for both ThunQ and hand-crafted queries can be explained by a combination



(a) Average end-to-end latency (b) Breakdown of ThunQ latency

Fig. 9. Latency in function of policy attribute count.



(a) Average end-to-end latency (b) Breakdown of ThunQ latency

Fig. 10. Latency in function of the number of tenants.

of the extra work required to check extra attributes in the query and request pagination in Spring Data, which generates extra count queries.

Q2. Next, we investigate the impact of the number of tenants on the end-to-end latency. We increased the number of tenants by adding brokers that are each assigned 1000 documents. We also enforced the policy that“*A broker can only view the documents that are assigned to the broker*”. Adding new brokers impacts two dimensions of the system. First, The size of the database increases, as each broker is assigned a fixed number of records. Second, the authorization policy becomes narrower, as the ratio between the records that the broker is authorized to view and the total number of records decreases. As before, application requests are paged with 50 records per page.

Fig. 10a shows the impact of the number of brokers in the system on the end-to-end latency. ThunQ closely follows the performance of hand-crafted queries, with the latency of both techniques increasing for a larger number of tenants. As shown earlier in Q1, policy selectivity only has a limited impact on the latency of either fine-grained authorization systems. This implies that the increase in latency can mostly be attributed to the increase in database size. The latency of the postfilter increases sharply once the system exceeds 10 tenants. This increase is mostly likely caused by the increase in policy selectivity. The behavior of the postfilter in Fig. 8a confirms this observation. The performance breakdown of ThunQ's end-to-end latency in Fig. 10b shows that the end-to-end latency is dominated by the database operations of the account-state service. This implies that relative overhead of ThunQ decreases as the number of tenants increases, which makes ThunQ better suited to protect applications with larger databases.

Discussion. Our results indicate that the impact of policy selectivity, attribute count, and the number of tenants on the performance of ThunQ is similar to the impact of these parameters on the performance of hand-crafted queries. However, postfiltering outperforms both approaches in scenarios where policies are broad and have a high attribute count. Nonetheless, ThunQ exhibits better performance characteristics for multi-tenant applications, such as e-insurance, that have to support numerous tenants with narrow policies, while still offering the flexibility required by policy customization. We did not consider the use of database indexes which might greatly enhance ThunQ's performance.

As discussed in Section 3, thunks are forwarded in their entirety between microservices to ensure loose coupling. Although this approach can cause thunks to contain policies that are not required by downstream services, we can assume that this overhead is relatively small for two reasons. First, thunks are composed of residual policies, which often reduces the size of the thunks. Second, generalizing our evaluation results, we can assume that the cost of query execution will be the dominant source of overhead in most target systems.

5 Related Work

This section first presents work related to access control for databases, followed by a discussion of security techniques for microservice applications.

Access Control for Databases. Enforcing access control at the level of database records is a non trivial problem. Next, we provide an overview of some techniques proposed by literature for fine-grained access control in database systems.

FGAC [24] enforces authorization policies on individual database records by defining a set of *authorization views* that restrict access to the database. Authorization views scale well to large result sets, but they break separation of concerns between security administration and application development, as authorization views are defined in the database's native query language. Moreover, FGAC scales poorly in terms of administrative overhead. FGAC represents each subject by a separate database user, which not only causes significant administrative overhead but is also problematic for multi-tenant applications, which often integrate with the IAM systems of their tenants.

Bouncer [16] aims to scale fine-grained access control with respect to large groups of users. It does so by inserting an enforcement point between the database and the application The enforcement point first performs an authorization check when a query arrives at the database. The result set of this query is then passed back to Bouncer, which uses a postfilter to exclude any unauthorized records. However, postfiltering does not scale well for large result sets [3].

Sequoia [3] combines the strengths of FGAC and Bouncer by rewriting database queries based on XACML policies. This approach results in low latency enforcement of expressive policies, even in systems with a large number of users. However, Sequoia does not provide an end-to-end solution for access control in applications with distributed application logic and data, such as multi-tenant microservice applications. Moreover, Sequoia instances receive policy updates individually, such that there are no guarantees that multiple Sequoia instances enforce a consistent set of policies on a single distributed control flow.

Securing Microservices. Securing microservice applications [11,19] is challenging, and it requires a holistic approach at different layers of the software stack for in-depth defense. Next, we discuss some security techniques which are put forward by literature to secure microservice applications.

Access control ensures that only authorized entities can interact with the protected system. Most solutions for application-level access control [9,15,20,29] either enforce policies within a single application domain [9,29] or in a setting with multiple parties [15,20]. To ensure interoperability, most solutions use standardized technologies, such as OAuth [15,29], UMA [20] and XACML [15]. The aforementioned systems enforce access control on the level of application requests, while ThunQ also enforces fine-grained policies at the data-record level.

Access control can also be enforced at the network level [21,31,37], either by leveraging Software Defined Networks (SDNs) [31], application containers [37], or a combination of both SDNs and the Host Identity Protocol (HIP) [21].

Managing authorization policies in microservice is challenging due to the multitude of services and the complexity of their interactions. One solution is to mine policies from historical application data [36] and install them at the application services. *AutoArmor* [14] offers a more holistic approach, as it extracts policies from the microservice code and keeps the policies up-to-date.

Application-level access control, such as ABAC, can leak sensitive information about its users. TSAP [38] is a system that is designed to protect the users' attributes by assigning attribute sensitivity and resource server trust levels.

Monitoring and Anomaly Detection aims to completely mediate and monitor application requests [31]. Recent work leverages anomaly detection to detect suspicious behavior through microservice RPC calls [7] or circumvent attacks against auto-scaling infrastructure by identifying cyclic patterns in application load [22].

Deception techniques aim to confuse attackers by setting up decoys and traps in the microservice application. Sandnet [17] leverages SDNs and CRIU (Checkpoint/Restore In Userspace) to create a sandboxed environment for suspicious application containers that are possibly compromised by an attacker.

Moving Target Defense (MTD) targets to reduce an attack's economy of scale by introducing variation in the microservice application. The challenge of MTD is selecting the appropriate variation technique to increase the resiliency of the application in a trade-off between security and performance. Recent work proposes to use vulnerability rating systems such as ORRM (OWASP Risk Rating Methodology) and CVSS (Common Vulnerability Scoring System) to select the appropriate variations [33]. Alternatively, MTD can use custom metrics such as *betweenness centrality* [13] to choose the most suitable variation technique.

A Trusted Execution Environment (TEE), such as Intel Secure Guard Extensions (SGX), is another technique to protect microservice applications. Squad [30] leverages TEEs for the secure delivery of application secrets and critical system configuration parameters. *Vert.x Vault* [4] extends the Eclipse *Vert.x* framework for microservices with secure application components that protect specific parts of the application using TEEs.

Integrity Protection aims to protect the integrity of artifacts and configuration of microservice applications from insider threats. Protecting the integrity of these systems often requires a combination of security techniques, such as *remote attestation*, *access control*, and *audit* [1]. Integrity protection can be used to ensure part of ThunQ's trust requirements presented in Section 3.

The discussion above highlights some of the techniques available for securing microservices. Even though ThunQ is able to efficiently enforce access control, it should be used in tandem with other security techniques.

6 Conclusion and Future Work

This work presented ThunQ, a distributed authorization middleware for multi-tenant microservice applications. ThunQ ensures data confidentiality by denying unauthorized requests as soon as possible and enforcing authorization policies *lazily*. ThunQ uses *partial policy evaluation* to make authorization decisions early at the *API gateway* and piggybacks the resulting *residual policies* as a *thunk* on the application request. This scheme moves the policies close to the data that is required to evaluate them, keeping the sensitive records within their local microservice context.

Our evaluation shows that ThunQ's performance is suitable to support large-scale multi-tenant microservice applications. ThunQ has limited overhead and performs better than postfiltering at large scales. Moreover, ThunQ's performance is comparable to the baseline hand-crafted implementation.

As a part of future work, we want to support authorization policies that use data from multiple data-sources for policy evaluation, for example by means of the *Command Query Responsibility Segregation* [23] pattern for microservices. Another effort can be focused on supporting obligations and HBAC policies [5].

References

1. Ahmadvand, M., Pretschner, A., Ball, K., Eyring, D.: Integrity protection against insiders in microservice-based infrastructures: From threats to a security framework. In: STAF. Springer (2018)

2. Bertino, E., Sandhu, R.: Database security-concepts, approaches, and challenges. IEEE TDSC **2**(1) (2005)
3. Bogaerts, J., Lagaisse, B., Joosen, W.: Sequoia: A middleware supporting policy-based access control for search and aggregation in data-driven applications. IEEE TDSC **18**(1) (2021)
4. Brenner, S., Hundt, T., Mazzeo, G., Kapitza, R.: Secure cloud micro services using intel sgx. In: DIAS. Springer (2017)
5. Brewer, D., Nash, M.: The chinese wall security policy. In: Proc. IEEE S&P 1989 (1989)
6. Bystr, C., Heyman, J., Hamrén, J., Heyman, H., Holmberg, L.: Locust. `https://locust.io/`
7. Chen, J., Huang, H., Chen, H.: Informer: Irregular traffic detection for containerized microservices rpc in the real world. In: Proc. SEC'19. ACM (2019)
8. De Win, B., Piessens, F., Joosen, W., Verhanneman, T.: On the importance of the separation-of-concerns principle in secure software engineering. In: ACSAC - WAEPSSD (2003)
9. Faravelon, A., Chollet, S., Verdier, C., Front, A.: Configuring private data management as access restrictions: From design to enforcement. In: ICSOC 2012. Springer (2012)
10. Guo, C.J., Sun, W., Huang, Y., Wang, Z.H., Gao, B.: A framework for native multi-tenancy application development and management. In: CEC-EEE (2007)
11. Hannousse, A., Yahiouche, S.: Securing microservices and microservice architectures: A systematic mapping study. Comput. Sci. Rev. **41** (2021)
12. Hu, V., Ferraiolo, D., Kuhn, R., Schnitzer, A., Sandlin, K., Miller, R., Scarfone, K.: Guide to attribute based access control (abac) definition and consideration. Tech. rep., NIST (2014)
13. Jin, H., Li, Z., Zou, D., Yuan, B.: Dseom: A framework for dynamic security evaluation and optimization of mtd in container-based cloud. IEEE TDSC **18**(3) (2021)
14. Li, X., Chen, Y., Lin, Z., Wang, X., Chen, J.H.: Automatic policy generation for inter-service access control of microservices. In: USENIX Security 21. USENIX Association (2021)
15. Nehme, A., Jesus, V., Mahbub, K., Abdallah, A.: Fine-grained access control for microservices. In: FPS. Springer (2019)
16. Opyrchal, L., Cooper, J., Poyar, R., Lenahan, B., Daniel, Z.: Bouncer: Policy-based fine grained access control in large databases. IJSIA **5**(2) (2011)
17. Osman, A., Bruckner, P., Salah, H., Fitzek, F.H.P., Strufe, T., Fischer, M.: Sandnet: Towards high quality of deception in container-based microservice architectures. In: IEEE ICC (2019)
18. Parducci, B., Lockhart, H.: extensible access control markup language (xacml) version 3.0. Standard, OASIS (2013)
19. Pereira-Vale, A., Fernandez, E.B., Monge, R., Astudillo, H., Márquez, G.: Security in microservice-based systems: A multivocal literature review. Comput. Secur. **103** (2021)
20. Preuveneers, D., Joosen, W.: Towards multi-party policy-based access control in federations of cloud and edge microservices. In: IEEE Euro S&PW (2019)
21. Ranjbar, A., Komu, M., Salmela, P., Aura, T.: Synaptic: Secure and persistent connectivity for containers. In: IEEE/ACM CCGRID (2017)
22. Ravichandiran, R., Bannazadeh, H., Leon-Garcia, A.: Anomaly detection using resource behaviour analysis for autoscaling systems. In: NetSoft and Workshops (2018)

23. Richardson, C.: Microservices Patterns. Manning Publications Co. (2018)
24. Rizvi, S., Mendelzon, A., Sudarshan, S., Roy, P.: Extending query rewriting techniques for fine-grained access control. In: Proc. SIGMOD '04. ACM (2004)
25. Samarati, P., de Vimercati, S.C.: Access control: Policies, models, and mechanisms. In: FOSAD. Springer (2001)
26. Sandall, T.: Partial evaluation, `https://blog.openpolicyagent.org/partial-evaluation-162750eaf422`
27. Sandhu, R.S.: Lattice-based access control models. Computer **26**(11) (1993)
28. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. Computer **29**(2) (1996)
29. ShuLin, Y., JiePing, H.: Research on unified authentication and authorization in microservice architecture. In: IEEE ICCT (2020)
30. da Silva, M.S.L., de Oliveira Silva, F.F., Brito, A.: Squad: A secure, simple storage service for sgx-based microservices. In: LADC (2019)
31. Sun, Y., Nanda, S., Jaeger, T.: Security-as-a-service for microservices-based cloud applications. In: IEEE CloudCom (2015)
32. Taibi, T., Lenarduzzi, V., Pahl, C.: Architectural patterns for microservices: A systematic mapping study. In: Proc. CLOSER. SciTePress (2018)
33. Torkura, K.A., Sukmana, M.I., Kayem, A.V., Cheng, F., Meinel, C.: A cyber risk based moving target defense mechanism for microservice architectures. In: IEEE BDCloud (2018)
34. Verhanneman, T., Piessens, F., De Win, B., Joosen, W.: Uniform application-level access control enforcement of organizationwide policies. In: ACSAC '05 (2005)
35. Westkämper, T., Dijkstra, R., Tims, J., Bain, R.: Querydsl. `http://www.querydsl.com/`
36. Xu, Z., Stoller, S.D.: Mining attribute-based access control policies. IEEE TDSC **12**(5) (2015)
37. Zaheer, Z., Chang, H., Mukherjee, S., Van der Merwe, J.: Eztrust: Network-independent zero-trust perimeterization for microservices. In: Proc. SOSR'19. ACM (2019)
38. Zhang, G., Liu, J., Liu, J.: Protecting sensitive attributes in attribute based access control. In: ICSOC 2012 Workshops. Springer (2013)
39. Keycloak. `https://www.keycloak.org/`
40. Rego. `https://www.openpolicyagent.org/docs/latest/policy-language/`
41. Open policy agent. `https://www.openpolicyagent.org/`
42. Spring boot. `https://spring.io/projects/spring-boot`
43. Spring data. `https://spring.io/projects/spring-data`
44. Spring cloud gateway. `https://spring.io/projects/spring-cloud-gateway`
45. Zuul. `https://github.com/Netflix/zuul`

ThunQ: A Distributed and Deep Authorization Middleware for Early and Lazy Policy Enforcement in Microservice Applications
Resubmission of paper #7

Abstract. Online software services are often designed as multi-tenant, API-based, microservice architectures. However, sharing service instances and storing sensitive data in a shared data store causes significant security risks. Application-level access control plays a key role in mitigating this risk by preventing unauthorized access to the application and data. Moreover, a microservice architecture introduces new challenges for access control on online services, as both the application logic and data are highly distributed. First, unauthorized requests should be denied as soon as possible, preferably at the facade API. Second, sensitive data should stay in the context of its microservice during policy evaluation. Third, the set of policies enforced on a single application request should be consistent for the entire distributed control flow.

To solve these challenges, we present ThunQ, a distributed authorization middleware that enforces authorization policies both early at the facade API, as well as lazily by postponing authorization decisions to the appropriate data context. To achieve this, ThunQ leverages two techniques called partial evaluation and query rewriting, which support policy enforcement both at the facade API, as well as deep in the data tier.

We implemented and open-sourced ThunQ as a set of reusable components for the Spring Cloud and Data ecosystem. Experimental results in an application case study show that ThunQ can efficiently enforce authorization policies in microservice applications, with acceptable increases in latency as the number of tenants and access rules grow.

1 Introduction

Contemporary online services often provide a customer-facing API and adopt an internal architecture based on application-level multi-tenancy and microservices. Application-level multi-tenancy [10], as illustrated in Fig. 1, benefits from economies of scale by sharing resources between the tenants, such as the application and database. However, storing sensitive tenant data poses significant security risks. Application-level access control [34] is a key security technique that mitigates these risks by enforcing authorization policies at the application-level to block unauthorized access to resources. Moreover, multi-tenant applications require that both the application provider and tenants can specify these policies. In particular, the provider specifies the basic authorization policies for the platform, while the tenants can provide additional policies that further restrict

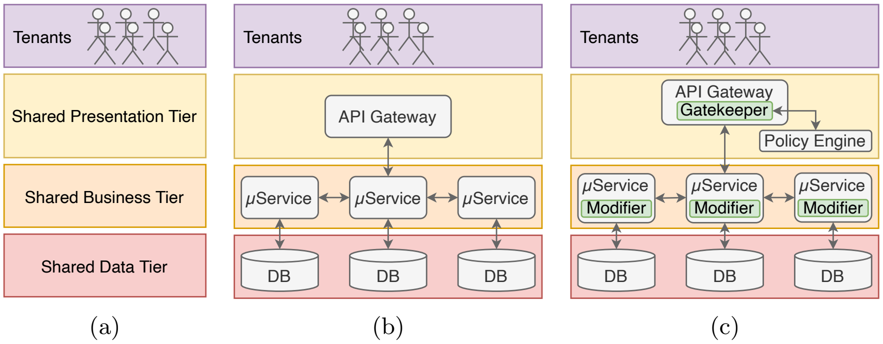


Fig. 1. Overview of application-level multi-tenancy (a) for both microservice applications (b) and applications with ThunQ (c). ThunQ's components are shown in green.

access by their end-users to comply with internal authorization policies. For example a tenant policy may state that: *"An insurance company employee can only view insurance documents of customers that are assigned to the employee."*

Supporting tenant specific policies requires an appropriate level of modularity, separation of concerns and adaptation of the related software artefacts [8]. While single-tenant applications can embed the authorization logic directly in the database query to enforce fine-grained access control, it is no longer feasible for multi-tenant applications with custom authorization policies per tenant. Custom policies require a more flexible approach where policies can be updated at run-time, as new tenants are continuously added to the application.

A frequently used architectural pattern to realize multi-tenant applications are *microservices* [23]. Microservice applications often adopt the *API gateway* [32] and the *database-per-service* [32] pattern as shown in Fig. 1b. The distribution of application logic and data in multi-tenant microservice applications introduces the following new challenges for access control in such applications:

1. Unauthorized requests should be denied *as soon as possible* (ASAP), such that unauthorized resource usage and control flows in the distributed microservice application are minimized.
2. Sensitive data should stay in the context of its microservice during policy evaluation, i.e. data from the data tier should not flow to the API gateway when evaluating authorization policies.
3. The set of policies enforced on a single application request should be consistent for the entire distributed control flow, as policies are no longer only enforced at the facade API but throughout the entire application.

Existing work on application-level access control [15,18,20,29,34] and API gateways [29,44,45] aims to enforce authorization policies ASAP, resulting in a permit or deny. However, these solutions require that sensitive data is brought outside of its microservice context. Other related work focuses on enforcing access control in application databases [3,16,24]. These solutions aim to restrict access by enforcing fine-grained authorization policies on the data records by either rewriting the original database query [3], defining authorization views [24] or by filtering database records after retrieving them from the database [16]. However, securing database access is only a part of the challenges to enforce a consistent set of authorization policies over a large number of microservices.

To address the challenges and shortcomings above, we present *ThunQ*, a distributed authorization middleware for multi-tenant microservice applications designed to efficiently and consistently enforce a set of authorization policies on distributed application services and data. ThunQ enforces authorization policies early in the distributed control flow, as well as deep down in the *data tier*. ThunQ achieves this by adding the *gatekeeper*, *policy engine* and *query modifier* components to the generic microservice architecture as shown in Fig. 1c. The gatekeeper and policy engine use partial policy evaluation [26] to create *thunks* that are piggybacked on the application request. The thunks are then used by the query modifier to enforce authorization policies deep in the data tier.

We implemented and open-sourced ThunQ as a set of reusable components for the Spring Cloud and Data ecosystem. Our evaluation shows that ThunQ performs notably better than state-of-practice postfiltering approaches. Moreover, ThunQ's overhead is largely independent of the number of application tenants and the complexity of the tenant specific policies.

The remainder of this text is structured as follows. Section 2 presents the motivational use case and provides the reader with background on access control and ThunQ's supporting technologies. Section 3 presents the architecture and the security model of the ThunQ middleware. Section 4 discusses the evaluation and results. Section 5 discusses related work and Section 6 concludes this work.

2 Motivational Use Case and Background

This section presents the motivation and background for ThunQ. We start with presenting *e-insurance*, an anonymized industrial case study of a multi-tenant insurance brokering platform with a microservice architecture and API-based online service offering. Next, we discuss background on access control models and ThunQ's enabling technologies.

The E-Insurance Case Study. In the financial industry, insurance companies or insurers do not always sell their insurance products directly to end customers. Instead, they employ intermediaries, called insurance brokers, to bring their products to the customer. Brokers negotiate insurance contracts with the customers and take care of the paperwork related to the contract. Furthermore, customers should have access to information regarding their insurance products, such as the current balance of their life insurance account. As shown in Fig. 2, e-insurance integrates insurers, brokers, and customers into a single platform that shares their insurance documents. E-insurance is responsible for storing the insurance contracts and their related documents, as well as offering advanced

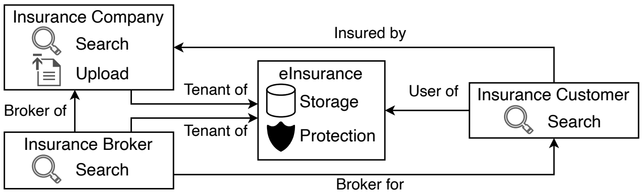


Fig. 2. Participants of the e-insurance application.

search operations on stored documents. However, as the contents of the insurance documents are sensitive, the results of the search operations should only include the information which the user is authorized to view.

Access Control Analysis. Ensuring the confidentiality of the insurance documents is the primary security goal of e-insurance. To achieve confidentiality, e-insurance must restrict access to only those users who are authorized to access a given document. Whether or not a user is authorized to access a document is determined by *authorization policies.* E-insurance defines two sets of policies: platform policies which are specified by e-insurance itself, and tenant policies, which are specified by the tenants to further restrict access by their end-users. Next, we provide a sample of possible policies.

P1. (platform) Brokers can only view documents assigned to them.
P2. (platform) Customers can only view documents that belong to them.
P3. (broker) Only senior employees can view documents worth over $100k.
P4. (insurer) Employees can only view the documents assigned to them.
P5. (insurer) Employees can only view documents during working hours.

Challenges. Given the discussion above, we can identify the following challenges for e-insurance. First, the application must guarantee the confidentiality of insurance documents by enforcing both platform and tenant policies. Second, e-insurance must offer the performance necessary to support numerous tenants and documents. Searching documents should be fast even as the number of tenants and documents increases. Finally, the set of policies applied to a single application request should be consistent for the entire distributed control flow.

Background. Access control models are models that determine which subjects, such as users and processes, are authorized to access a given object, such as files and other resources. The choice of access control model has a significant impact on the kind of authorization policies that can be expressed. Examples of access control models include Lattice Based Access Control [27] and Role Based Access Control [28]. We focus on Attribute-Based Access Control (ABAC) [12] in combination with Policy-Based Access Control (PBAC) [25]. ABAC models access rights by assigning attributes to the subjects and objects. ABAC makes authorization decisions dynamically, based on the assigned attributes and the environment, such as location and time. PBAC, on the other hand, makes decisions based on authorization policies. These policies are evaluated by a policy engine that uses an access control model, such as the attributes and context assigned by the ABAC model, to reach an authorization decision.

The separation of concerns between authorization policies and the mechanism to enforce them is a key principle in secure software engineering [8]. PBAC [25] decouples policy from mechanism by using policy engines to evaluate policies written in authorization policy languages. The *Open Policy Agent* (OPA) [41] is a policy engine that supports the *Rego* [40] policy language for writing policies. Rego policies use the attributes provided by the authorization request, as well as the access control model stored by OPA. OPA supports both full and *partial evaluation* [26] of authorization policies. Partial evaluation reduces a given policy by substituting the known variables in the policy and evaluating the involved expressions. The result of a partial evaluation is a reduced version of the original policy that only contains unknown variables. We further refer to the reduced version of the policy as the *residual policy*.

The OASIS eXtensible Access Control Markup Language (XACML) [18] is an industry standard for access control. XACML provides a specification for the XACML policy language and a reference architecture for authorization systems. XACML combines PBAC and ABAC, using XML documents to specify authorization policies. The XACML reference architecture contains the following components: (i) a Policy Enforcement Point (PEP), which intercepts incoming application requests, (ii) a Policy Administration Point (PAP), that manages the system's policies, (iii) a Policy Information Point (PIP), that stores the access control attributes, and (iv) a Policy Decision Point (PDP), which takes authorization decisions based on the context provided by the PAP and PIP.

3 ThunQ Middleware

This section presents ThunQ, a distributed authorization middleware for multi-tenant microservice applications. ThunQ is designed to efficiently enforce a consistent set of authorization policies on distributed application services and data. ThunQ combines *partial policy evaluation* [26] and *query rewriting* [2,3] to enforce authorization policies both *early* and *lazily*. Early enforcement denies unauthorized requests as soon as possible, while lazy enforcement pushes access decisions further down the distributed control flow. Next, we define ThunQ's security model, followed by a description of the architecture and its key elements.

Security Model. Fig. 1b depicts the system model for applications supported by ThunQ. ThunQ assumes that all application requests pass through an API gateway [32], which is a *facade* for the services in the *business tier*. Microservices in the business tier execute the actual business logic of the application and can call other microservices. Additionally, the services in the business tier rely on the databases in the *data tier* for persistence. ThunQ supports dedicated databases per service, as well as a single database that is shared between microservices. Given this system model, ThunQ makes the following trust assumptions.

A1 All services shown in Fig. 1b are trusted and operate correctly.

A2 Policies defined by the platform's security administrators are correct, meaning that they enforce the intended security policies.
A3 Tenant policies do not impact existing security properties of the system, i.e. policies are defined by the provider's security consultant after a requirements analysis of the tenant.
A4 Security administrators are trusted, i.e. there is no insider threat caused by the security staff.

The primary security goal of ThunQ is to restrict access to the distributed application logic and data by enforcing platform and tenant policies. First, ThunQ should deny unauthorized requests as soon as possible. Second, ThunQ should enable the confidentiality of application data by enforcing the authorization policies on individual data records deep in the data tier. ThunQ only achieves these goals when the following assumptions about the attacker hold.

A5 An attacker can only interact with the system through the APIs provided by the platform.
A6 An attacker cannot impersonate any other user.
A7 The attacker has no access to side-channels in the communication between the system and the attacker.

ThunQ's Overall Architecture. The authorization architecture of ThunQ is shown in Fig. 3. ThunQ adds the following components to realize its security goals. First, ThunQ adds the *gatekeeper* to the API gateway. The gatekeeper performs authorization checks and piggybacks the *thunks* on the application request. Second, ThunQ transparently adds a *query modifier* to the microservices. The modifier intercepts database queries from the application and rewrites them to enforce authorization policies. Next, we discuss the application request flow with distributed policy evaluation, followed by ThunQ's core architectural elements.

Distributed Policy Evaluation. Policy evaluation in ThunQ is distributed, early and lazy. Evaluation is distributed, as ThunQ evaluates policies at different points in the microservice application, early, as unauthorized requests are denied ASAP by partial evaluation, and lazy, as ThunQ postpones access decisions by piggybacking the residual policies to the appropriate data context. More specifically, policy evaluation in ThunQ starts at the API gateway where incoming application requests are intercepted by the gatekeeper (1). The gatekeeper then inspects the request and extracts any information regarding the subject. Next, the gatekeeper selects the policies applicable to the request and calls the *policy engine* with the subject information and the selected policies as arguments (2). The policy engine then partially evaluates the policies and returns the *residual policies* to the gatekeeper (3). The gatekeeper transforms the residual policies into a thunk and attaches the thunk to the application request. Alternatively, the policy engine returns a deny, in which case the gateway blocks the request.

Next, the API gateway forwards the request to the relevant microservice (4.1). The microservice then handles the request either by querying the database

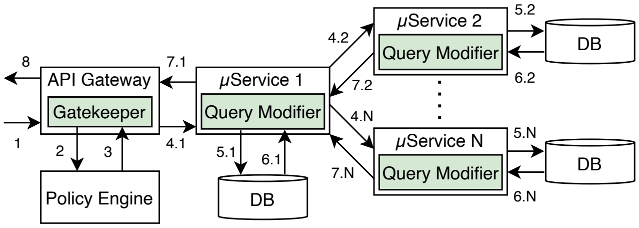
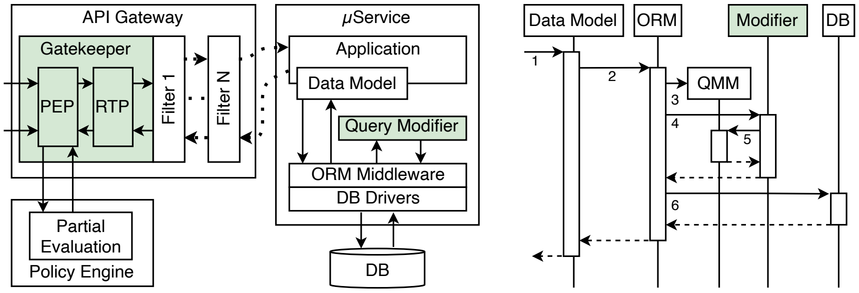


Fig. 3. Authorization architecture. ThunQ's components are shown in green.



(a) Components

(b) Query execution flow

Fig. 4. Detailed view of ThunQ's interactions with the application components.

(5.1 - 6.1) or by calling other microservices and piggybacking the thunk (4.x - 7.x). Each query made by the application gets intercepted by the query modifier, where the query gets rewritten to enforce the authorization policies before being passed to the database (5.1). The result of the rewritten query is then sent back to the application (6.1). After the data is retrieved, the application can perform other operations, eventually finishing the request and replying to the caller (7.1). Eventually, the API gateway receives the response and forwards it to the client (8). Note that the same rewriting procedure (5.x - 6.x) is applied when the service calls other microservices to handle the request.

We next discuss the core architectural elements of the ThunQ middleware. The ThunQ middleware consists of two main components the gatekeeper and the query modifier. These components and a policy engine are added transparently to the microservice application as shown in Fig. 4.

Gatekeeper. The gatekeeper enforces the authorization policies on the requests both early and lazily. As depicted in Fig. 4a, the gatekeeper is attached to the API gateway as a filter component that intercepts all incoming application requests. The gatekeeper can be further broken down into the *Policy Enforcement Point* or PEP, and the *Request Transformation Point* or RTP. The PEP is a modified version of a XACML PEP [18] and is responsible for sending requests for partial policy evaluation to the policy engine. The policy engine responds

```
1  allow {                            allow {
2    user.tenant=="insurer"             doc.tenant_id==67
3    doc.tenant_id==user.tenant_id      doc.employee_id==42
4    user.role=="account_manager"     }
5    doc.employee_id==user.id
6  }
```

Fig. 5. Example policy (left) and the residual policy after partial evaluation (right).

with either a set of residual policies or a deny. In the case of a deny, the PEP blocks the application request, denying the request early. Alternatively, the policy engine responds with a residual policy, in which case the PEP sends the residual policies to the RTP, which transforms the residual policies into Boolean expressions and adds the expressions to the thunk. The RTP is a new component in the XACML dataflow that is responsible for augmenting application requests, in particular by attaching a thunk for lazy enforcement.

Fig. 5 shows an example of partial policy evaluation at the gateway. The policy consists of rules which are defined by the provider at lines 2 and 3, as well as by the tenant at lines 4 and 5. Note that all subject attributes are available at the gateway such that lines 2 and 4 can be evaluated and, if necessary, denied early. This while lines 3 and 5 must be evaluated lazily in the data tier, as the attributes of *doc* are not accessible from the current evaluation context.

We realized ThunQ's gatekeeper as a *gateway filter* instance for *Spring Cloud Gateway* [44]. The gateway filter is implemented as a *stateless* instance to minimize ThunQ's memory footprint. However, the concept of the gatekeeper is more general and is not limited to this specific software implementation. The policy engine is provided by *Open Policy Agent* (OPA) [41], as it supports partial policy evaluation. OPA can be deployed as either a standalone service or a sidecar of the API gateway, depending on its memory consumption. For e-insurance we deployed OPA as a stateless sidecar, as memory use was limited to 10 MiB.

Thunks. A thunk is the key data structure that enables lazy and consistent policy evaluation in a distributed control flow. Thunks are created by the RTP which transforms the residual policies forwarded by the PEP into Boolean expressions. These expressions are added to a thunk by the RTP and piggybacked on the request. By piggybacking the thunks, the residual policies are able to travel together with distributed control flow, where they can be used by other ThunQ components to enforce fine-grained authorization policies deep in the data tier. As shown in Fig. 6, a thunk is a collection of *URL path selectors* mapped to a Boolean expression. The selectors are used by the query modifier to determine which residual policies are relevant for the intercepted database query. To ensure loose coupling, thunks are forwarded in their entirety between microservices. Note that each application request is processed with a consistent set of policies, as the same thunk is re-used for the entire the distributed control flow.

Query Modifier. The query modifier rewrites database queries such that the queries enforce authorization policies on individual data records. Note that the

```
{
  "/accountStates/*":"doc.tenant_id=67 && doc.employee_id=42",
  "/hospitalBills/*": <BoolExpr#2>,
  "/*": <BoolExpr#3>
}
```

Fig. 6. Example of a thunk encoding the partial policy of Fig. 5 and others.

```
SELECT *               SELECT *
FROM account_states    FROM account_states
                       WHERE tenant_id=67 AND employee_id=42
                         AND <BoolExpr#3>
```

Fig. 7. Example of query rewriting by the query modifier. The original query on the left is rewritten using the thunk in Fig. 6 with */accountStates/all* as request path.

query modifier only augments search queries since these operate on large result sets. As shown in Fig. 4a, the query modifier is attached to the application as a plugin for the *Object Relational Mapper(ORM) middleware.* ORMs often provide hooks that enable third-party extensions to modify database queries through the *query meta-model* (QMM).

To rewrite queries, the query modifier must first determine the relevant residual policies to enforce. These policies are encoded as Boolean expressions in the thunks that are piggybacked on the application requests. The relevant Boolean expressions are selected by matching the URL path selectors of the thunk against the application request path. The matching expressions are then joined using a conjunction to create a Boolean expression that encodes all the matched residual policies at once. This expression is then woven into the meta-model of the database query by adding the expression to the *predicate* of the query's model. The modified query then gets further processed by the ORM middleware before it is sent to the database. The result of the query then is sent back to the ORM without passing through the modifier. An example of the effect of query rewriting on a SQL query is illustrated in Fig. 7.

Fig. 4b shows the flow of a database query in detail. First, the application invokes a search method on the data model (1). Next, the data model contacts the ORM middleware (2) which creates a query meta-model that corresponds to the method call (3). This meta-model is an internal representation of the query that the ORM will map later to a database specific query. Next, the ORM passes the meta-model to the query modifier (4), which rewrites the query as described earlier using the meta-model (5). After calling the modifier, the ORM instantiates the actual database query using the modified meta-model (6) and returns the result back to the data model. ThunQ's query modifier was realized as a component for the *Spring Data* [43] ORM middleware. The query modifier utilizes the Querydsl [35] query meta-model to rewrite database queries. Furthermore, the query modifier is implemented as a stateless component to minimize ThunQ's memory footprint.

4 Evaluation

This section discusses the evaluation of the ThunQ middleware with a key focus on the performance overhead of the middleware solution. We compare ThunQ against two alternative approaches for fine-grained authorization in the data tier, namely *postfiltering* [16] and *hand-crafted queries*. Postfiltering enforces authorization policies on data queries by checking each record in the result set against a policy engine. Hand-crafted queries, on the other hand, encode the authorization policies directly in the application queries. Although the last approach is impractical for multi-tenant applications, it represents the best-case scenario for query-based approaches to enforce fine-grained authorization, as it doesn't have the overhead of ThunQ's middleware components. The evaluation aims to answer the following questions related to multi-tenancy and performance.

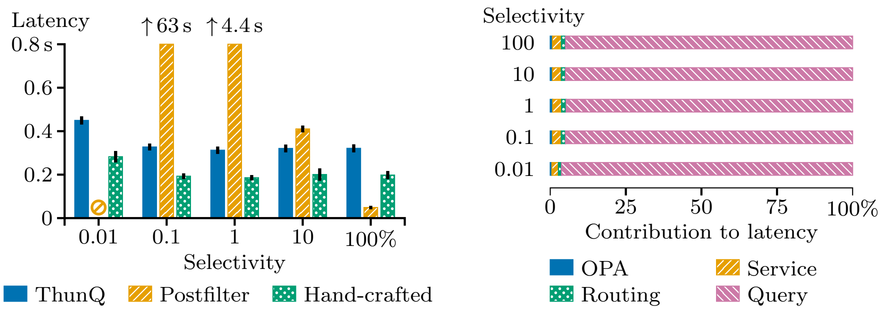
Q1 What is the impact of the properties of the enforced policies on the latency? As tenants specify policies that further restrict access by their end-users, it decreases the number of records included in the results. Also, adding policies can increase the number of attributes required for evaluation.
Q2 What is the impact on end-to-end latency when the number of tenants grows? As microservice applications are very sensitive to increases in latency, the overhead of ThunQ should not put limitations on the number of tenants.

Evaluation Setup. All experiments were performed on a proof-of-concept application (PoC) that is based on the e-insurance case study discussed in Section 2. The PoC was deployed in an AKS Kubernetes cluster in the Microsoft Azure public cloud. The Kubernetes control plane was hosted on a single Standard_B2s VM with 2 CPUs and 4GiB of memory, while the PoC runs inside a node pool consisting of 3 Standard_D4as_v4 VMs with 4 CPUs and 16GiB of memory. To simulate application users, we used the Locust [6] load generation tool.

The PoC consists of the following services: an *API gateway*, an *account-state service*, a *datastore*, and an *IAM* system. The API gateway is an instance of Spring Cloud Gateway [44] with an additional *gatekeeper* filter as discussed in Section 3. The account-state service handles statements of account balances generated by life insurances. The service is realized a Spring Boot [42] application augmented with the *query modifier* from Section 3. Furthermore, the datastore is an instance of Azure SQL and the IAM system is provided by Keycloak [39].

Q1. We first investigate the impact of two policy properties called *policy selectivity* and *attribute count*. Policy selectivity is the ratio between the number of data records still included after applying the policy to the result set and the size of the original result set. Policies with low values for selectivity are called *narrow*, as only a small portion of the original result set is included. Policies with high selectivity values are called *broad* as more records remain included. The attribute count of a policy, on the other hand, defines how many attributes are required by a policy for lazy evaluation.

We configured the experiments as follows. Clients send requests through the API gateway to fetch data from the account-state service, which has a database



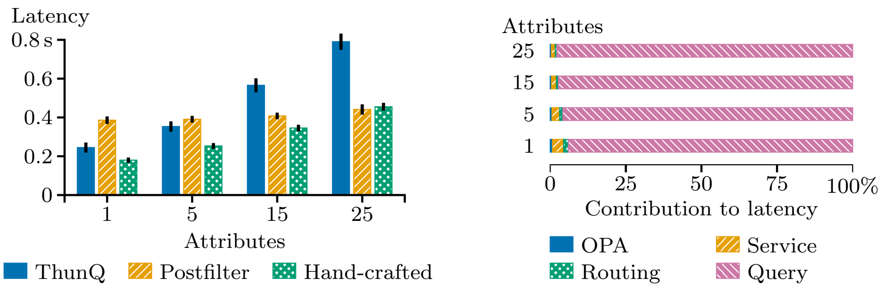
(a) Average end-to-end latency (b) Breakdown of ThunQ latency

Fig. 8. Latency in function of policy selectivity.

with 1 million records. Application requests are paginated and retrieve only the first 50 accessible records that satisfy the authorization policies. The policies in both scenarios were synthetically generated to show the impact of the different policy properties. The policies for the experiments with varying policy selectivity only have a single attribute, while the experiments with varying attribute count have policies with a selectivity of 10%.

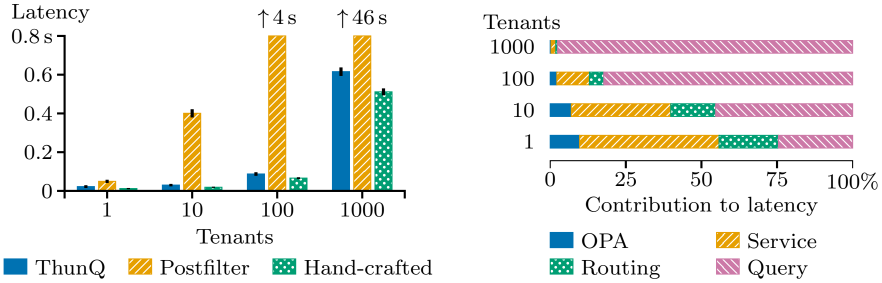
Impact of Policy Selectivity. Fig. 8a shows the impact of policy selectivity on the end-to-end latency. For ThunQ and hand-crafted queries, latencies are largely unaffected by policy selectivity, with only a minor increase for very narrow policies. In addition, the breakdown of the ThunQ's request latency shown in Fig. 8b, indicates that ThunQ's latency is dominated by the database query. The results for postfiltering show low latencies for policies with selectivity between 10 and 100%. This is a consequence of paged requests, as filling a page requires that only a limited number of records have to be checked against the policy engine. In contrast, narrow policies have high latencies. The decrease in selectivity means that more database records need to be checked by the policy engine before a single page can be filled, in turn increasing the overhead of the postfilter and the overall latency. A final observation concerns the results for policies with a selectivity of 100%. In this case, postfiltering outperforms both ThunQ and hand-crafted queries. This is caused by the way Spring Data handles request paging for ThunQ and hand-crafted queries.

Impact of Attribute Count. Fig. 9 shows the relation between the number of attributes used in the lazy evaluation of a policy and the end-to-end request latency for policies with a 10% selectivity. All three fine-grained authorization methods show a linear increase in latency for higher attribute counts. Although postfiltering initially performs worse than the other techniques, its slope is less steep compared to ThunQ or hand-crafted queries. Consequently it matches or outperforms the other solutions for higher attribute counts. The steeper slope for both ThunQ and hand-crafted queries can be explained by a combination



(a) Average end-to-end latency (b) Breakdown of ThunQ latency

Fig. 9. Latency in function of policy attribute count.



(a) Average end-to-end latency (b) Breakdown of ThunQ latency

Fig. 10. Latency in function of the number of tenants.

of the extra work required to check extra attributes in the query and request pagination in Spring Data, which generates extra count queries.

Q2. Next, we investigate the impact of the number of tenants on the end-to-end latency. We increased the number of tenants by adding brokers that are each assigned 1000 documents. We also enforced the policy that "*A broker can only view the documents that are assigned to the broker*". Adding new brokers impacts two dimensions of the system. First, The size of the database increases, as each broker is assigned a fixed number of records. Second, the authorization policy becomes narrower, as the ratio between the records that the broker is authorized to view and the total number of records decreases. As before, application requests are paged with 50 records per page.

Fig. 10a shows the impact of the number of brokers in the system on the end-to-end latency. ThunQ closely follows the performance of hand-crafted queries, with the latency of both techniques increasing for a larger number of tenants. As shown earlier in Q1, policy selectivity only has a limited impact on the latency of either fine-grained authorization systems. This implies that the increase in latency can mostly be attributed to the increase in database size. The latency of the postfilter increases sharply once the system exceeds 10 tenants. This increase is mostly likely caused by the increase in policy selectivity. The behavior of the postfilter in Fig. 8a confirms this observation. The performance breakdown of ThunQ's end-to-end latency in Fig. 10b shows that the end-to-end latency is dominated by the database operations of the account-state service. This implies that relative overhead of ThunQ decreases as the number of tenants increases, which makes ThunQ better suited to protect applications with larger databases.

Discussion. Our results indicate that the impact of policy selectivity, attribute count, and the number of tenants on the performance of ThunQ is similar to the impact of these parameters on the performance of hand-crafted queries. However, postfiltering outperforms both approaches in scenarios where policies are broad and have a high attribute count. Nonetheless, ThunQ exhibits better performance characteristics for multi-tenant applications, such as e-insurance, that have to support numerous tenants with narrow policies, while still offering the flexibility required by policy customization. We did not consider the use of database indexes which might greatly enhance ThunQ's performance.

As discussed in Section 3, thunks are forwarded in their entirety between microservices to ensure loose coupling. Although this approach can cause thunks to contain policies that are not required by downstream services, we can assume that this overhead is relatively small for two reasons. First, thunks are composed of residual policies, which often reduces the size of the thunks. Second, generalizing our evaluation results, we can assume that the cost of query execution will be the dominant source of overhead in most target systems.

5 Related Work

This section first presents work related to access control for databases, followed by a discussion of security techniques for microservice applications.

Access Control for Databases. Enforcing access control at the level of database records is a non trivial problem. Next, we provide an overview of some techniques proposed by literature for fine-grained access control in database systems.

FGAC [24] enforces authorization policies on individual database records by defining a set of *authorization views* that restrict access to the database. Authorization views scale well to large result sets, but they break separation of concerns between security administration and application development, as authorization views are defined in the database's native query language. Moreover, FGAC scales poorly in terms of administrative overhead. FGAC represents each subject by a separate database user, which not only causes significant administrative overhead but is also problematic for multi-tenant applications, which often integrate with the IAM systems of their tenants.

Bouncer [16] aims to scale fine-grained access control with respect to large groups of users. It does so by inserting an enforcement point between the database and the application The enforcement point first performs an authorization check when a query arrives at the database. The result set of this query is then passed back to Bouncer, which uses a postfilter to exclude any unauthorized records. However, postfiltering does not scale well for large result sets [3].

Sequoia [3] combines the strengths of FGAC and Bouncer by rewriting database queries based on XACML policies. This approach results in low latency enforcement of expressive policies, even in systems with a large number of users. However, Sequoia does not provide an end-to-end solution for access control in applications with distributed application logic and data, such as multi-tenant microservice applications. Moreover, Sequoia instances receive policy updates individually, such that there are no guarantees that multiple Sequoia instances enforce a consistent set of policies on a single distributed control flow.

Securing Microservices. Securing microservice applications [11,19] is challenging, and it requires a holistic approach at different layers of the software stack for in-depth defense. Next, we discuss some security techniques which are put forward by literature to secure microservice applications.

Access control ensures that only authorized entities can interact with the protected system. Most solutions for application-level access control [9,15,20,29] either enforce policies within a single application domain [9,29] or in a setting with multiple parties [15,20]. To ensure interoperability, most solutions use standardized technologies, such as OAuth [15,29], UMA [20] and XACML [15]. The aforementioned systems enforce access control on the level of application requests, while ThunQ also enforces fine-grained policies at the data-record level.

Access control can also be enforced at the network level [21,31,37], either by leveraging Software Defined Networks (SDNs) [31], application containers [37], or a combination of both SDNs and the Host Identity Protocol (HIP) [21].

Managing authorization policies in microservice is challenging due to the multitude of services and the complexity of their interactions. One solution is to mine policies from historical application data [36] and install them at the application services. *AutoArmor* [14] offers a more holistic approach, as it extracts policies from the microservice code and keeps the policies up-to-date.

Application-level access control, such as ABAC, can leak sensitive information about its users. TSAP [38] is a system that is designed to protect the users' attributes by assigning attribute sensitivity and resource server trust levels.

Monitoring and Anomaly Detection aims to completely mediate and monitor application requests [31]. Recent work leverages anomaly detection to detect suspicious behavior through microservice RPC calls [7] or circumvent attacks against auto-scaling infrastructure by identifying cyclic patterns in application load [22].

Deception techniques aim to confuse attackers by setting up decoys and traps in the microservice application. Sandnet [17] leverages SDNs and CRIU (Checkpoint/Restore In Userspace) to create a sandboxed environment for suspicious application containers that are possibly compromised by an attacker.

Moving Target Defense (MTD) targets to reduce an attack's economy of scale by introducing variation in the microservice application. The challenge of MTD is selecting the appropriate variation technique to increase the resiliency of the application in a trade-off between security and performance. Recent work proposes to use vulnerability rating systems such as ORRM (OWASP Risk Rating Methodology) and CVSS (Common Vulnerability Scoring System) to select the appropriate variations [33]. Alternatively, MTD can use custom metrics such as *betweenness centrality* [13] to choose the most suitable variation technique.

A Trusted Execution Environment (TEE), such as Intel Secure Guard Extensions (SGX), is another technique to protect microservice applications. Squad [30] leverages TEEs for the secure delivery of application secrets and critical system configuration parameters. *Vert.x Vault* [4] extends the Eclipse *Vert.x* framework for microservices with secure application components that protect specific parts of the application using TEEs.

Integrity Protection aims to protect the integrity of artifacts and configuration of microservice applications from insider threats. Protecting the integrity of these systems often requires a combination of security techniques, such as *remote attestation*, *access control*, and *audit* [1]. Integrity protection can be used to ensure part of ThunQ's trust requirements presented in Section 3.

The discussion above highlights some of the techniques available for securing microservices. Even though ThunQ is able to efficiently enforce access control, it should be used in tandem with other security techniques.

6 Conclusion and Future Work

This work presented ThunQ, a distributed authorization middleware for multi-tenant microservice applications. ThunQ ensures data confidentiality by denying unauthorized requests as soon as possible and enforcing authorization policies *lazily*. ThunQ uses *partial policy evaluation* to make authorization decisions early at the *API gateway* and piggybacks the resulting *residual policies* as a *thunk* on the application request. This scheme moves the policies close to the data that is required to evaluate them, keeping the sensitive records within their local microservice context.

Our evaluation shows that ThunQ's performance is suitable to support large-scale multi-tenant microservice applications. ThunQ has limited overhead and performs better than postfiltering at large scales. Moreover, ThunQ's performance is comparable to the baseline hand-crafted implementation.

As a part of future work, we want to support authorization policies that use data from multiple data-sources for policy evaluation, for example by means of the *Command Query Responsibility Segregation* [23] pattern for microservices. Another effort can be focused on supporting obligations and HBAC policies [5].

References

1. Ahmadvand, M., Pretschner, A., Ball, K., Eyring, D.: Integrity protection against insiders in microservice-based infrastructures: From threats to a security framework. In: STAF. Springer (2018)

2. Bertino, E., Sandhu, R.: Database security-concepts, approaches, and challenges. IEEE TDSC **2**(1) (2005)
3. Bogaerts, J., Lagaisse, B., Joosen, W.: Sequoia: A middleware supporting policy-based access control for search and aggregation in data-driven applications. IEEE TDSC **18**(1) (2021)
4. Brenner, S., Hundt, T., Mazzeo, G., Kapitza, R.: Secure cloud micro services using intel sgx. In: DIAS. Springer (2017)
5. Brewer, D., Nash, M.: The chinese wall security policy. In: Proc. IEEE S&P 1989 (1989)
6. Bystr, C., Heyman, J., Hamrén, J., Heyman, H., Holmberg, L.: Locust. `https://locust.io/`
7. Chen, J., Huang, H., Chen, H.: Informer: Irregular traffic detection for containerized microservices rpc in the real world. In: Proc. SEC'19. ACM (2019)
8. De Win, B., Piessens, F., Joosen, W., Verhanneman, T.: On the importance of the separation-of-concerns principle in secure software engineering. In: ACSAC - WAEPSSD (2003)
9. Faravelon, A., Chollet, S., Verdier, C., Front, A.: Configuring private data management as access restrictions: From design to enforcement. In: ICSOC 2012. Springer (2012)
10. Guo, C.J., Sun, W., Huang, Y., Wang, Z.H., Gao, B.: A framework for native multi-tenancy application development and management. In: CEC-EEE (2007)
11. Hannousse, A., Yahiouche, S.: Securing microservices and microservice architectures: A systematic mapping study. Comput. Sci. Rev. **41** (2021)
12. Hu, V., Ferraiolo, D., Kuhn, R., Schnitzer, A., Sandlin, K., Miller, R., Scarfone, K.: Guide to attribute based access control (abac) definition and consideration. Tech. rep., NIST (2014)
13. Jin, H., Li, Z., Zou, D., Yuan, B.: Dseom: A framework for dynamic security evaluation and optimization of mtd in container-based cloud. IEEE TDSC **18**(3) (2021)
14. Li, X., Chen, Y., Lin, Z., Wang, X., Chen, J.H.: Automatic policy generation for inter-service access control of microservices. In: USENIX Security 21. USENIX Association (2021)
15. Nehme, A., Jesus, V., Mahbub, K., Abdallah, A.: Fine-grained access control for microservices. In: FPS. Springer (2019)
16. Opyrchal, L., Cooper, J., Poyar, R., Lenahan, B., Daniel, Z.: Bouncer: Policy-based fine grained access control in large databases. IJSIA **5**(2) (2011)
17. Osman, A., Bruckner, P., Salah, H., Fitzek, F.H.P., Strufe, T., Fischer, M.: Sandnet: Towards high quality of deception in container-based microservice architectures. In: IEEE ICC (2019)
18. Parducci, B., Lockhart, H.: extensible access control markup language (xacml) version 3.0. Standard, OASIS (2013)
19. Pereira-Vale, A., Fernandez, E.B., Monge, R., Astudillo, H., Márquez, G.: Security in microservice-based systems: A multivocal literature review. Comput. Secur. **103** (2021)
20. Preuveneers, D., Joosen, W.: Towards multi-party policy-based access control in federations of cloud and edge microservices. In: IEEE Euro S&PW (2019)
21. Ranjbar, A., Komu, M., Salmela, P., Aura, T.: Synaptic: Secure and persistent connectivity for containers. In: IEEE/ACM CCGRID (2017)
22. Ravichandiran, R., Bannazadeh, H., Leon-Garcia, A.: Anomaly detection using resource behaviour analysis for autoscaling systems. In: NetSoft and Workshops (2018)

23. Richardson, C.: Microservices Patterns. Manning Publications Co. (2018)
24. Rizvi, S., Mendelzon, A., Sudarshan, S., Roy, P.: Extending query rewriting techniques for fine-grained access control. In: Proc. SIGMOD '04. ACM (2004)
25. Samarati, P., de Vimercati, S.C.: Access control: Policies, models, and mechanisms. In: FOSAD. Springer (2001)
26. Sandall, T.: Partial evaluation, `https://blog.openpolicyagent.org/partial-evaluation-162750eaf422`
27. Sandhu, R.S.: Lattice-based access control models. Computer **26**(11) (1993)
28. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. Computer **29**(2) (1996)
29. ShuLin, Y., JiePing, H.: Research on unified authentication and authorization in microservice architecture. In: IEEE ICCT (2020)
30. da Silva, M.S.L., de Oliveira Silva, F.F., Brito, A.: Squad: A secure, simple storage service for sgx-based microservices. In: LADC (2019)
31. Sun, Y., Nanda, S., Jaeger, T.: Security-as-a-service for microservices-based cloud applications. In: IEEE CloudCom (2015)
32. Taibi, T., Lenarduzzi, V., Pahl, C.: Architectural patterns for microservices: A systematic mapping study. In: Proc. CLOSER. SciTePress (2018)
33. Torkura, K.A., Sukmana, M.I., Kayem, A.V., Cheng, F., Meinel, C.: A cyber risk based moving target defense mechanism for microservice architectures. In: IEEE BDCloud (2018)
34. Verhanneman, T., Piessens, F., De Win, B., Joosen, W.: Uniform application-level access control enforcement of organizationwide policies. In: ACSAC '05 (2005)
35. Westkämper, T., Dijkstra, R., Tims, J., Bain, R.: Querydsl. `http://www.querydsl.com/`
36. Xu, Z., Stoller, S.D.: Mining attribute-based access control policies. IEEE TDSC **12**(5) (2015)
37. Zaheer, Z., Chang, H., Mukherjee, S., Van der Merwe, J.: Eztrust: Network-independent zero-trust perimeterization for microservices. In: Proc. SOSR'19. ACM (2019)
38. Zhang, G., Liu, J., Liu, J.: Protecting sensitive attributes in attribute based access control. In: ICSOC 2012 Workshops. Springer (2013)
39. Keycloak. `https://www.keycloak.org/`
40. Rego. `https://www.openpolicyagent.org/docs/latest/policy-language/`
41. Open policy agent. `https://www.openpolicyagent.org/`
42. Spring boot. `https://spring.io/projects/spring-boot`
43. Spring data. `https://spring.io/projects/spring-data`
44. Spring cloud gateway. `https://spring.io/projects/spring-cloud-gateway`
45. Zuul. `https://github.com/Netflix/zuul`