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

Anonymous Authors

No Institute Given

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 access control both early at the facade API, as well as lazily by postponing access 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 access control 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 [7], 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 [25] is a key security technique that mitigates these risks by enforcing security 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 security policies. In particular, the provider specifies the basic security policies for the platform,

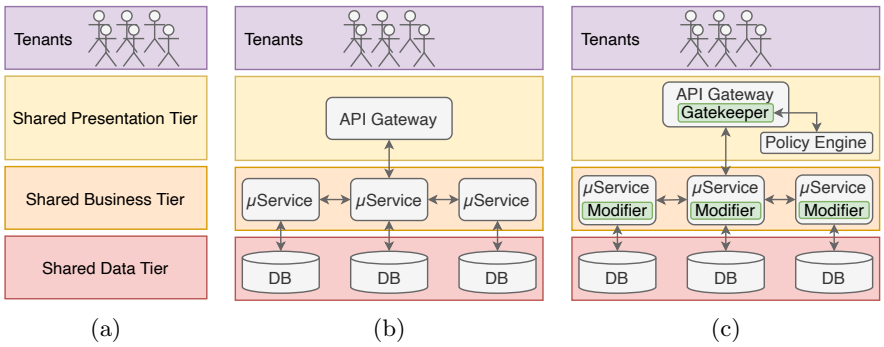


Fig. 1. Overview of application-level multi-tenancy (a) for both microservice applications (b) and applications with ThunQ (c). ThunQ's components are shown in green.

while the tenants can provide additional policies that further restrict access by their end-users to comply with corporate security 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 [5]. While single-tenant applications can embed the access control logic directly in the database query to enforce fine-grained access control, it is no longer feasible for multi-tenant applications with custom security policies per tenant. Custom policies require a more flexible approach where policies can be updated at runtime, as new tenants are continuously added to the application.

A frequently used architectural pattern to realize multi-tenant applications are *microservices* [15]. Microservice applications often adopt the *API gateway* [24] and the *database-per-service* [24] 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 security 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 [27, 25] and API gateways [22, 30] aims to enforce access control 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 [16, 2]. These solutions aim to restrict access by enforcing fine-grained access control on the data records by either rewriting the original database query [2], defining authorization views [16] or by filtering database records after retrieving them from the database [13]. However, securing database access is only a part of the challenges to enforce a consistent set of access control 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 access control policies on distributed application services and data. ThunQ enforces access control 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 [18] to create *thunks* that are piggybacked on the application request. The thunks are then used by the query modifier to enforce access control 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

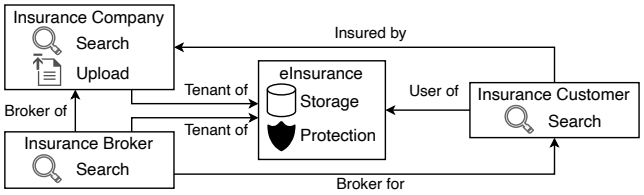


Fig. 2. Participants of the e-insurance application.

insurance contracts and their related documents, as well as offering advanced 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.

Security 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 *security 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 security 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 security 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 security policies that can be expressed. Examples of access control models include Lattice Based Access Control [19] and Role Based Access Control [20]. This work focuses on Attribute-Based Access Control (ABAC) [10] in combination with Policy-Based Access Control (PBAC) [17]. ABAC models access rights by assigning attributes to the subjects and objects. ABAC makes access decisions dynamically, based on the assigned attributes and the environment, such as location and time. PBAC, on the other hand, makes authorization

decisions based on access control 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 security policies and the mechanism to enforce them is a key principle in secure software engineering [5]. PBAC [17] decouples policy from mechanism by using policy engines to evaluate access control policies written in access control policy languages. The *Open Policy Agent* (OPA) [12] is a policy engine that supports the *Rego* [14] 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* [18] of access control 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) [27] 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 security 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 access control policies on distributed application services and data. ThunQ combines *partial policy evaluation* [18] and *query rewriting* [1, 2] to enforce access control 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 [24], 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 access control 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 security 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 access control policies. Next, we discuss the application request flow with distributed policy evaluation, followed 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

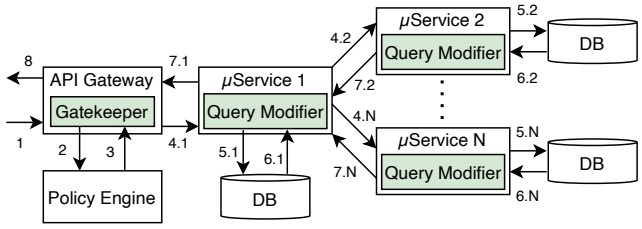
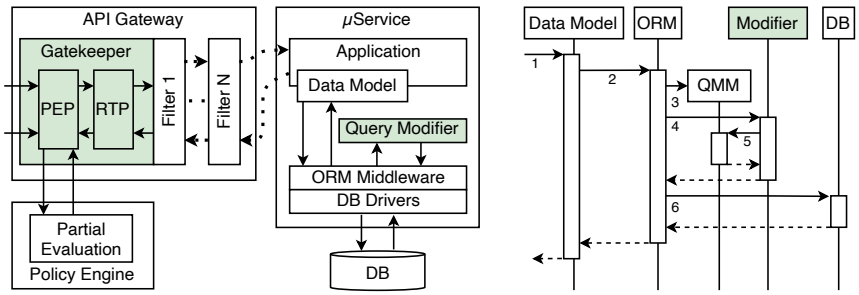


Fig. 3. Security architecture. ThunQ's components are shown in green.



(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 access control 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 access control 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 [27] 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 security context.

We realized ThunQ's gatekeeper as a *gateway filter* instance for *Spring Cloud Gateway* [22]. 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) [12], as it supports partial policy evaluation.

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 access control 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. 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 access control policies on individual data records. Note that the 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).

```
{
  "/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.

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* [23] ORM middleware. The query modifier utilizes the Querydsl [26] query meta-model to rewrite database queries.

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 access control in the data tier, namely *postfiltering* [13] and *hand-crafted queries*. Postfiltering enforces access

control 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 access control 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 access control, 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:

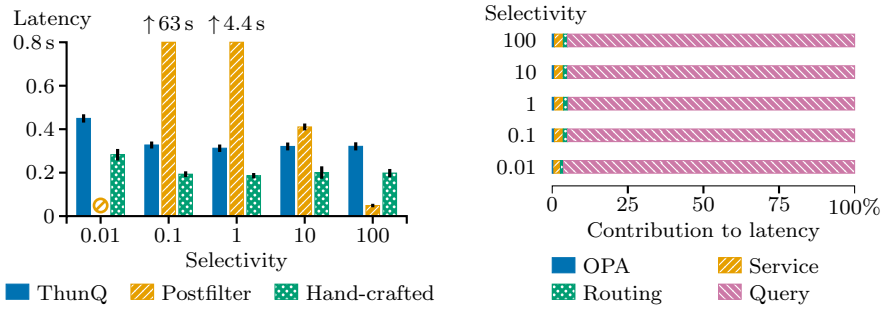
Q1 What is the impact of the properties of the enforced policies on the latency? As tenants specify policies that further restrict access by their end-users, it decreases the number of records included in the results. Also, adding policies can increase the number of attributes required for evaluation.
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 [4] 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 [22] 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 [21] 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 [11].

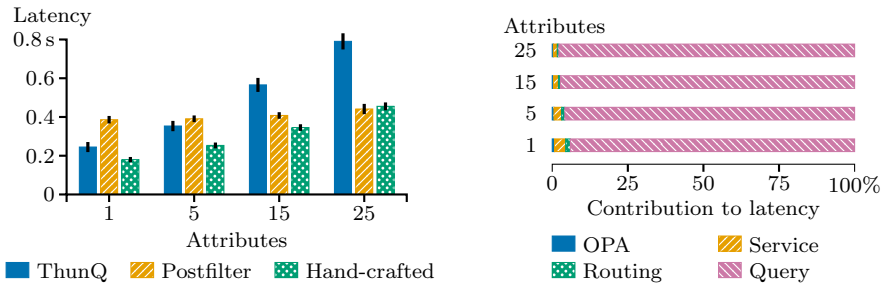
Q1. We first investigate the impact of two policy properties called *policy selectivity* and *attribute count*. Policy selectivity is the ratio between the number of data records still included after applying the policy to the result set and the size of the original result set. Policies with low values for selectivity are called *selective*, as only a small portion of the original result set is included. Policies with high selectivity values are called *permissive* 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 with 1 million records. Application requests are paginated and retrieve only the first 50 accessible records that satisfy the access control policies. The access control 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%



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

Fig. 8. Latency in function of policy selectivity.

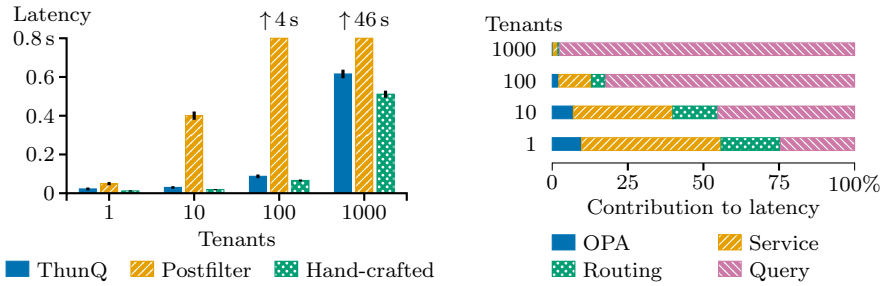


(a) Average end-to-end latency (b) Breakdown of ThunQ latency

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

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 highly selective 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, highly selective policies have high latencies. The increase 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



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

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

latency for policies with a 10% selectivity. All three fine-grained access control 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 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 access 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 access control policy becomes more selective, 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 access control 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 permissive 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 highly selective policies while still offering the flexibility required by policy customization. We also did not consider the use of database indexes, these indexes exploit extra domain knowledge and might greatly enhance ThunQ's performance.

5 Related Work

We briefly introduced related work on access control in Section 2. Next, we discuss the remainder of related work on access control.

Fine-Grained Access Control (FGAC) [16] enforces access control policies on individual database records by defining a set of *authorization views* that restrict access to the database. Although FGAC scales well to large result sets, it has the following problems. First, FGAC defines the authorization views in the database's native query language, which breaks the separation of concerns between security administration and application development. Second, FGAC does represent each subject by a separate database user, which does not scale well a large number of users. Additionally, representing subjects by database users is problematic for multi-tenant applications, as these applications often integrate with the IAM system of their tenants.

Bouncer [13] 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 the query arrives at the database. Next, bouncer uses postfiltering to exclude any unauthorized database records from the query response. However, postfiltering does not scale well for large result sets [2].

Sequoia [2] combines the strengths of FGAC and Bouncer by rewriting database queries based on XACML policies. This results in low latency enforcement with the ability to enforce expressive policies for 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.

OAuth [8] uses *access tokens* that contain user attributes for authorization. Like access tokens, thunks are attached to the application request. However, unlike access tokens, thunks contain *residual policies* instead of attributes.

Prior work on restricting access to sensitive data in service-oriented computing [6] uses *secure proxies* to protect services. However, access decisions of secure proxies are coarse-grained, as they permit or deny entire application requests.

In addition to application data, user attributes also require adequate protection. The *TSAP* [29] system exposes attributes to resource providers based on the sensitivity level of the attribute and the trust level of the resource provider.

While ABAC policies offer many advantages, it can be challenging to migrate to ABAC from a legacy access control system, such as RBAC [20] or Access Control Lists [9]. *Policy mining* [28] is an automated solution that transforms the legacy access control model to ABAC policies and attributes.

6 Conclusion and Future Work

This work introduced ThunQ, a distributed authorization middleware for multi-tenant microservice applications. ThunQ ensures data confidentiality by denying unauthorized requests as soon as possible and enforcing security policies *lazily*.

ThunQ uses *partial policy evaluation* to make access control 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 access control policies that use data from multiple data-sources for policy evaluation, for example by means of the *Command Query Responsibility Segregation* [15] pattern for microservices. Another effort can be focused on supporting obligations and HBAC policies [3].

References

[1] E. Bertino and R. Sandhu. "Database Security-Concepts, Approaches, and Challenges". In: *IEEE TDSC* 2.1 (2005).

[2] J. Bogaerts, B. Lagaisse, and W. Joosen. "SEQUOIA: A Middleware Supporting Policy-Based Access Control for Search and Aggregation in Data-Driven Applications". In: *IEEE TDSC* 18.1 (2021).

[3] D.F.C. Brewer and M.J. Nash. "The Chinese Wall security policy". In: *Proc. IEEE S&P 1989*.

[4] C. Bystr et al. *Locust*. URL: `https://locust.io/`.

[5] B. De Win et al. "On the importance of the separation-of-concerns principle in secure software engineering". In: *ACSAC - WAEPSSD 2003*.

[6] A. Faravelon et al. "Configuring Private Data Management as Access Restrictions: From Design to Enforcement". In: *ICSOC 2012*. Springer.

[7] C. J. Guo et al. “A Framework for Native Multi-Tenancy Application Development and Management”. In: *CEC-EEE 2007*.
[8] Dick Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. 2012.
[9] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. “Protection in Operating Systems”. In: *Commun. ACM* 19.8 (Aug. 1976).
[10] V. Hu et al. *Guide to Attribute Based Access Control (ABAC) Definition and Consideration*. Tech. rep. NIST, 2014.
[11] *Keycloak*. URL: `https://www.keycloak.org/`.
[12] *Open Policy Agent*. URL: `https://www.openpolicyagent.org/`.
[13] L. Opyrchal et al. “Bouncer: Policy-Based Fine Grained Access Control in Large Databases”. In: *IJSIA* 5.2 (2011).
[14] *Rego*. `https://www.openpolicyagent.org/docs/latest/policy-language/`. Accessed: 2021-05-26.
[15] C. Richardson. *Microservices Patterns*. Manning Publications Co., 2018.
[16] S. Rizvi et al. “Extending Query Rewriting Techniques for Fine-Grained Access Control”. In: *Proc. SIGMOD ’04*. ACM.
[17] P. Samarati and S. C. de Vimercati. “Access Control: Policies, Models, and Mechanisms”. In: *FOSAD 2001*. Springer.
[18] T. Sandall. *Partial Evaluation*. Feb. 2018. URL: `https://blog.openpolicyagent.org/partial-evaluation-162750eaf422` (visited on 05/12/2021).
[19] R. S. Sandhu. “Lattice-Based Access Control Models”. In: *Computer* 26.11 (Nov. 1993).
[20] R. S. Sandhu et al. “Role-Based Access Control Models”. In: *Computer* 29.2 (Feb. 1996).
[21] *Spring Boot*. URL: `https://spring.io/projects/spring-boot`.
[22] *Spring Cloud Gateway*. URL: `https://spring.io/projects/spring-cloud-gateway`.
[23] *Spring Data*. URL: `https://spring.io/projects/spring-data`.
[24] T. Taibi, V. Lenarduzzi, and C. Pahl. “Architectural Patterns for Microservices: A Systematic Mapping Study”. In: *Proc. CLOSER 2018*. SciTePress.
[25] T. Verhanneman et al. “Uniform application-level access control enforcement of organizationwide policies”. In: *ACSAC ’05*.
[26] T. Westkämper et al. *Querydsl*. URL: `http://www.querydsl.com/`.
[27] *eXtensible Access Control Markup Language (XACML) Version 3.0*. Standard. Jan. 2013.
[28] Z. Xu and S. D. Stoller. “Mining Attribute-Based Access Control Policies”. In: *IEEE TDSC* 12.5 (2015).
[29] G. Zhang, J. Liu, and J. Liu. “Protecting Sensitive Attributes in Attribute Based Access Control”. In: *ICSOC 2012 Workshops*. Springer.
[30] *Zuul*. URL: `https://github.com/Netflix/zuul`.