Re-Useable Access Control Plugin for Apollo Server

Klausing, Wilke
TU Berlin
klausing@campus.tu-berlin.de

Nguyen, Huy Viet
TU Berlin
huy.v.nguyen@campus.tu-berlin.de

***Abstract*—GraphQL is a query language for reading and mutating data in APIs. It provides the back-end developer a type system to describe a data schema. This in turn gives front-end developers of the API the power to request the exact data they need. Although GraphQL is perceived as 'the successor of the good old REST', there is no widely established way to easily implement access control in GraphQL APIs. In this paper we will give a deeper insight to GraphQL, touch on different access control models that are common practice, show existing technologies that enable access control in GraphQL and introduce a way to allow purposed-based access control in GraphQL Apollo Servers which goes beyond traditional, account- or role-based access control.**

***Index Terms*—Privacy engineering, access control, GraphQL, Apollo Server, data protection, purpose limitation, PBAC**

I. Introduction

The meaning of privacy and how it is defined is some of the most debated topic in information law today. There are multiple concepts and understanding of privacy. They all differ and there is no single valid definition of privacy since the concept and understanding differs due to various reasons such as cultures, disciplines, etc. That causes for example two different western cultures to have diverse concepts of privacy which was pointed out in the paper 'The Two Western Cultures of Privacy: Dignity Versus Liberty' by J. Whitman.

J. Whitman states that the US and UK think of privacy as *'liberty, and especially liberty against the state'* whereas continental Europe thinks of privacy as *'a form of protection of a right to respect and personal dignity'*. [1]

Although there exists not one single valid definition, privacy is still a relevant concept that is a major concern in today's information systems. Facebook paying a $5 billion fine to settle privacy concerns in 2019 [2] is one such example and shows the importance of privacy engineers in the tech industry. Privacy engineers have to ensure that privacy policies are respected and integrated into products. One way to ensure privacy is through access control, which restricts access to information depending on the user's authority. This paper is going to introduce a concept to create advanced access control in existing GraphQL APIs. (see chapter IV) The idea is to design a re-usable component that is easily to integrate into existing GraphQL frameworks and enables privacy engineers to implement access control beyond the scope of common models such as role-based access control. Before presenting the concept, this paper introduces background knowledge (see chapter II) by providing a brief overview of privacy engineering and digs deeper into access control, the most common models of access control and a query language named GraphQL that is used to create interfaces that enables information exchange between different applications. Afterwards, this paper will introduce existing technologies (see chapter III) by showcasing two technologies that enable the integration of basic access control into existing JavaScript GraphQL frameworks and lastly, present a re-usable, flexible and developer friendly access control plugin for Apollo servers.

II. Background

The goal of this chapter is to provide enough background knowledge to show the relevance of the plugin this paper is going to introduce.

A. Privacy Engineering

Privacy Engineering is an emerging discipline that addresses privacy and data protection in information systems and came to existence because of the gap between privacy research and actual practice. [3] The scientific paper 'Privacy Engineering: Shaping an Emerging Field of Research and Practice' by S. Gürses et al. defined privacy engineering as follows:

> *'Privacy engineering is an emerging research framework that focuses on designing, implementing, adapting, and evaluating theories, methods, techniques, and tools to systematically capture and address privacy issues in the development of sociotechnical systems.'* [3]

In recent years, the research community has made significant progress in theory and in research in privacy became relevant in all kinds of technology fields: in web standards [4], in cloud-based systems [5], in smart-grid systems [6] and much more.

Although research articles have spiked in the early 2000s and kept on growing (see figure 1), the amount of new articles regarding privacy engineering that can be found through Google Scholar has dropped steadily in recent years (see figure 2). That indicates a decline in active research although a recent paper 'A Shortage of Privacy Engineers' by L. Cranor and N. Sadeh state that there is a clear need for more privacy engineers and *'there are too few of these professionals today'* [7].

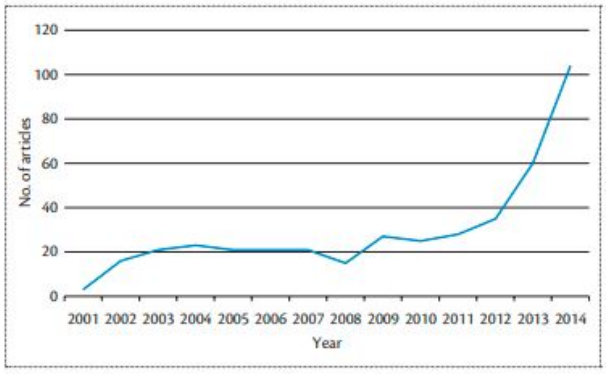


Fig. 1. The growing number of published privacy-engineering articles [3]

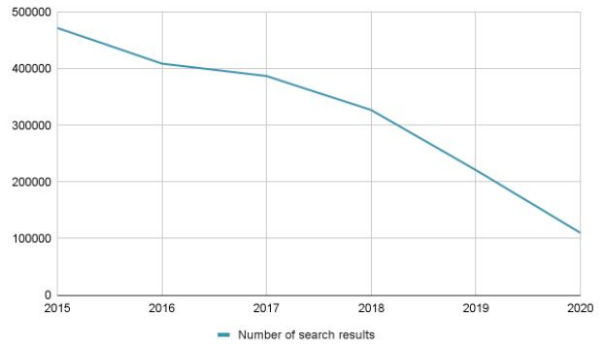


Fig. 2. Privacy Engineering search results on Google Scholar

B. Access Control (AC)

Privacy engineers need to be able to implement privacy control requirements into IT systems. In order to realize the requirements, access control plays a vital part. As the name already implies, access control allows a system to limit access to it's information or operations. It is used to constrain what a user can view, do directly or what other programs can execute on behalf of the user. This kind of constrain on the system plays an important role in computer systems and many papers mention the importance access control has:

> *'Access control is central to security in computer systems. Over the years, there have been many efforts to explain and improve access control, sometimes with logical ideas and tools.'* - M. Abadi [8]

> *'Access control is an indispensable part of any information sharing system.'* - H. Shen and P. Dewan [9]

Access control can be done in many ways and multiple models exist but they all follow a general concept. In access control we differentiate between subjects, objects, and access rights. Subjects are the users or programs who require some sort of access rights. Objects are the entities for which the subjects require the access rights. An entity could be a file, data or table. The access right defines which operation are allowed. A subject can only read, write, or delete a file if it has the access rights to do so. The process of distributing the access rights is done with access control models. It is an essential part of the security which can be implemented on different levels and ways. [10, p. 2-3] This paper is going to introduce the following access control models:

1) Rule-Based Access Control (RBAC or RB-RBAC)
2) Purpose-Based Access Control (PBAC)

C. Rule-Based Access Control (RBAC or RB-RBAC)

This model uses rules to decide if access is granted or not. Rules can be thought of as parameters, which need to be right. For instance, a rule could be that requests have to come from a specific location, IP address or port. If a malicious subject enters your network it still needs to find a way to match all the rules, which makes the network more secure. In contrast to Role-Based Access Control, Rule-Based Access Control does not rely on a subject to login with credentials. Therefore, it can be used in a broader context. The use of both models can also become handy, an example scenario would be if only logged subjects who use port 435 are allowed to access the files. Using both makes the access control a lot more flexible. [11]

D. Purpose-Based Access Control (PBAC)

The purpose based access control model is the attempt to build access control around the notion of purpose. A purpose is a description of reason to ask for access and the intended use of the data. Purposes can be distinguished between intended purpose and access purpose. Intended purpose refers to a description to the data, which describes how this data can be used. Access purpose, in contrast, refers to a purpose which needs to be send along with the request to access this data. How a purpose is determined can be challenging. An easy but unsafe way could be to trust an subject to describe its purpose correctly, therefore relying solely on trust. Another way would be to determine the purpose from a set of context data, like time or job function, to extract the real purpose. However, this would be hard to implement and possibly inefficient. After the purpose is determined, it can be matched with the intended purpose of the object and grand access. If a purpose is missing, access is not granted at all. Purpose-based access control can be used permissive and prohibitive. While prohibitive purpose access control tries to avoid to get access to data that is not wanted, permissive data access control does not grand access if an intended purpose is not part of the access purposes. Both can be implemented but is not necessarily required. Purposes are organized in a tree structure, called purpose tree. All nodes represent one purpose with the root as the most generalized purpose. When going down the tree, edges represent a specialization between nodes. This design allows an efficient storing and processing. [12, p. 3-4,6]

E. GraphQL

Access control is a core requirement for APIs and even more so when it comes to publicly exposed APIs. Hence Access Control is an important security technique that needs to be addressed in GraphQL which allows to implement APIs with query functionalities.

A mayor concept of GraphQL is to think about data not in terms of resource URLs or tables but rather as a graph of objects. The client formulates queries, mutations or subscriptions to the server. Queries are used to retrieve data from the server, whereas mutations are used to change data within a predefined schema. Subscriptions command the server to push a notification to the client if some specific data changes. While queries and mutations use HTTP protocol for data transfer, subscription requires a web socket with a constant connection between server and client. [13]

If a client wants to make a query request it needs to send an HTTP GET or POST request to the endpoint, which is usually called /graphql. Each request contains a query which looks similar to a JSON object. Filled fieldsi are used as search parameters to find the empty fields, which the client asked for. The fields are static types. In figure 3 you can see how a query can look like, with the respective answer on the right side.

Fig. 3. GraphQL Field example [14]

For GET requests, the query is encoded in the URL. This can look like the following example URL:

http://localhost:8080/graphql?query={...}*&*
variables={...}*& operation*={...}

When POST is used, the query is put into the request body.

On the server side, when receiving the requests, it needs to be parsed, processed and answered accordingly. Parsing and answering are defined in the GraphQL protocols and are greatly standardized. For these steps', libraries are widely available in many programming languages for different platforms. The process step, on the other hand, needs to be defined by the developer. Where the data comes from and what to include or exclude cannot be standardized because this highly depends on the use case. For each field that needs to be filled, one resolver is used. The resolver is basically a function to get the data. Where the data comes, is up the implementation of the resolver by the developer. It could, for instance, be a text file, database, or from a data object which is stored in memory. For a good performance are the resolvers executed in parallel, if possible, instead of sequentially. To avoid several requests to the same source, resolvers share resources between each other. [13]

To be able to execute queries another core concept of GraphQL is needed, which defines the capabilities of the API, called schema. The schema is a description of the server implementation. With the schema at hands the client knows how to formulate a request to the server. Each field of the schema is linked to one resolver. [15]

III. Existing Technologies

This chapter will introduce two APIs that are used for access control for current GraphQL frameworks. GraphQL Shield, which creates an additional permission layer on top of existing GraphQL APIs and GraphQL Filter which enables the back-end developer to filter the output of processed requests.

A. GraphQL Shield

GraphQL Shield is based on GraphQL Middleware which makes it possible to run arbitrary code before or after a GraphQL resolver is invoked. The Shield API was made to create field, type and role-based access control for any GraphQL and is compatible with all JavaScript GraphQL servers. The GraphQL Shield can be added into existing GraphQL Servers via GraphQL Middleware:

Listing 1. Integration [16]

```
// Permissions...

// Apply permissions middleware with applyMiddleware
// Giving any schema (instance of GraphQLSchema)

import { applyMiddleware } from 'graphql-middleware'
// schema definition...
schema = applyMiddleware(schema, permissions)
```

And rules can be set by using logical operations: OR, AND, NOT, CHAIN (rules will be executed one by one until one fails or all pass) and RACE (chain rules, execution stops once one of them returns true). An example of how role-based access control works with GraphQL Shield can be seen in Listing 2 .

Listing 2. Role-based Access Control [16]

```
import { shield, rule, and, or } from 'graphql-shield'

const isAdmin = rule()(async (parent, args, ctx, info) => {
  return ctx.user.role === 'admin'
})

const isEditor = rule()(async (parent, args, ctx, info) => {
  return ctx.user.role === 'editor'
})

const isOwner = rule()(async (parent, args, ctx, info) => {
  return ctx.user.items.some((id) => id === parent.id)
})

const permissions = shield({
  Query: {
    users: or(isAdmin, isEditor),
  },
  Mutation: {
    createBlogPost: or(isAdmin, and(isOwner, isEditor)),
  },
```

```
22    User: {
23      secret: isOwner,
24    },
25  })
```

The code in Listing 2 defines the following rules for a certain blog:

- Only an editor or an admin can see a list of all users
- Only an admin or an owner with an editor role can create blog posts
- Only the owner of the blog can see user's secrets

B. GraphQL Filter

GraphQL Filter is another API that was created based on GraphQL Middleware. It allows to set specific types private. The API is said to provide an additional privacy layer to GraphQL and replaces private information before the query result is sent back to the client. The API is the only framework apart from GraphQL Shield that provides access control for GraphQL as of now. There is no documentation available. There is one test case to showcase how to integrate GraphQL Filter into existing systems. The developer designed it in order to create a more flexible privacy control:

> 'It may be able to implement flexible read control with GraphQL-filter and GraphQL-shield.' [17]

No other extensions were integrateable APIs were found which shows the relevance of this paper's topic and a real need for a re-usable component that allows developers to easily add access control into their systems.

IV. Access Control Plugin

This chapter is going to give an overview of the plugin's functionality, the requirements that were defined to create a valuable re-usable plugin that can be used in a wide array of systems and how the configurations of the plugin are going to be.

A. Requirements

In order to create a practically valuable access control component and to make sure of the quality of the plugin, we are going to satisfy the requirements of F. Pallas et al. which were mentioned in the paper 'Towards Application-Layer Purpose-Based Access Control': [18]

1) **Attribute-level Access Control**: access control needs to be organized at attribute level
2) **Hierarchies of Purposes**: purposes need to be organized in a hierarchical structure to allow different levels of specificity when it comes to purposes
3) **Flexibility of Purpose Vocabulary**: the specification of purposes needs to flexible and changeable by every developer, so it can be integrated in a wide range of systems
4) **Distinction between Allowed & Prohibited Purposes**: there needs to be a distinction between permitted and prohibited purposes
5) **Database Independence**: the plugin won't make use of any kind of database to avoid incompatibilities
6) **Reasonable Performance Overhead**: the plugin needs to be as efficient and small as possible to not cause any large performance overhead in order to gain practical relevance
7) **Developer-Friendliness**: the plugin needs to be easy to understand and integrate into established architectural models and development stacks
8) **Re-usability**: to make the plugin easy to integrate in a wide range of systems

Each requirement will be discussed in the next following chapters and solutions for every requirement will be given.

B. Apollo Server

The Apollo Server is an open-source GraphQL Server that is community maintained. It is compatible with any GraphQL client and declares that:

> *'It's the best way to build a production-ready, self-documenting GraphQL API that can use data from any source.'* [19]

The framework is well documented and easy to setup. It is actively developed by over 400 contributors and is the second most used framework to create GraphQL Servers using JavaScript. The game changer and the reason the access control module is going to be built on top of Apollo Server is because Apollo Server allows the creation of plugins to extend server functionality. Plugins can perform custom operations in response to events that correspond to the GraphQL request life-cycle. The events our plugin is going to make use of are:

- **requestDidStart**: fires when a GraphQL request comes in - enables the plugin to check if the request comes with a purpose, whether it is a valid one and if rules for all fields/every request are fulfilled
- **willSendResponse**: fires before sending the GraphQL response - enables the plugin to filter information that can't be accessed based on the given purpose or header rules

Although creating a plugin for Apollo servers is going to limit the usage of the access control module to Apollo server systems, it is a good compromise since a plugin guarantees the re-usability and the easy integration of the module. (**Req. 8**) The performance overhead is also going to remain reasonable, since Apollo designed plugins in a way that the code is seamlessly added on top of the life-cycle events. (**Req. 6**)

C. Plugin Configuration

The plugin needs a flexible way of setting up different kinds of purposes and access control to allow any kind of system to use the plugin. In order to solve that requirement, the plugin is going to use two configuration files, so that developers can modify them at will and adjust the plugin for their needs (**Req. 3**). That also ensures that no database is necessary, which fulfills **Req. 5**. One configuration file enables developers to create and define different purposes ('*purpose.yml*'). The

second configuration file gives developers the ability to define acccess control rules ('*rules.json*').

D. Purpose Tree

The purpose tree of the plugin is created based in a YAML format which is easy to read (**Req. 7**). That allows purposes to be defined in a hierarchical structure (**Req. 2**). One such purpose tree can be seen in listing 3. In the example, the purpose tree consists of four main categories: 'research', 'marketing', 'health' and 'fitness'. If a rule allows access to a field for the purpose 'health', everything underneath it would automatically be accepted as well (f.e. diagnostic sleep analytics or predictive analytics). The purpose tree is highly configurable and can be changed based on the developer's needs.

Listing 3. Purpose Tree Example

```
purpose:
  research:
    – sleep research
    – cardiovascular research
  marketing:
    – personalized marketing
    – influencer marketing
  health:
    – descriptive analytics:
      – descriptive sleep analytics
      – descriptive cardiovascular analytics
    – diagnostic analytics:
      – diagnostic sleep analytics
      – diagnostic cardiovascular analytics
    – predictive analytics:
      – predictive sleep analytics
      – predictive cardiovascular analytics
    – prescriptive analytics:
      – prescriptive sleep analytics
      – prescriptive cardiovascular analytics
  fitness:
    – track activity
    – social fitness:
      – group challenge
```

E. Rule Creation

The second configuration file gives developers the ability to define access control rules. The rules can apply to any or all response fields (**Req. 1**) and are loaded at the initialization of the Apollo server and plugin. As of now, the plugin supports two different access control models:

1) Header-based access control: enables the user to create rules based on the request header information of the incoming GraphQL request
2) Purpose-based access control: enables the user to create rules based on the purpose given by incoming GraphQL requests

The rules configuration file uses the JSON format and allow the creation of header-based or purpose-based rules for allowing or prohibiting access to any or all given fields of a GraphQL request (**Req. 4**). The plugin is created in a way such that a rule is never processed more than once for incoming GraphQL requests, by storing validation results and re-using them if fields appear multiple times throughout a the response JSON.

An in-depth explanation of the rule structure of the two access control models and examples are shown on the GitHub Wiki [20].

V. Validation Use-Case

To validate our plugin we created a GraphQL API for Fitbit, a fitness tracker. The dataset is downloaded from Kaggle.com, a website which offers dataset for data science competitions. The dataset offers a lot of different features, like total steps, calories, or heartrate per seconds. Moreover, we added Person data which contains names, birthdays, and emails. For such a rich dataset it is easier to implement reasonable use-cases for our purposes. This scenario is going to showcase how to integrate the Access Control Plugin and validate how much performance overhead the plugin will cause by running the API with and without the plugin.

VI. Benchmarking

Benchmarking was done to asses and compare the performance of the Apollo server with and without the access control plugin. Three different tests were conducted, which differ in terms of the amount of data that is queried from the database. The first test returns a JSON Object with a total of 101.943 key/values, the second test returns 833 key/values and the third test returns 473 key/values. In each case, the procedure is the same. 1.000 requests are sent to the server and for each request the processing time is logged by the Apollo server itself through a logging plugin, which does a timestamp when a request comes in and before a response is send out to the client.

Two GCloud virtual machines (VM), one located in Germany and another one in central US were used as a testing environment. Both were setup identical and used the same type (machine type e2 with 2 vCPUs and 4 GB memory). The code is publicly accessible on Github [20].

The first test result of the German VM can be seen in figure 4 and figure 5 displays the results of the central US VM. The chart for the German VM shows more spread compared to the central US VM. Moreover, there is a striking increase in processing time at the end when access control was active, while the processing time increase within the central US VM stays at the same level.

The results for the second test are very similar, not just between the German and central US VM, but also in terms of performance when access control is active and not. The outliers especially at the starts can have their origin through the start up of the server, however the reason for the outliers throughout the test is unknown at this point.

The third and smallest test shows the same picture as for test two. Again there is a performance problem at the start, which could be caused by the server startup. It also has some outliers, which are above 10 ms processing time for which we cannot provide an answer yet.

Fig. 4. First Test (Germany)

Fig. 5. First Test (Central US)

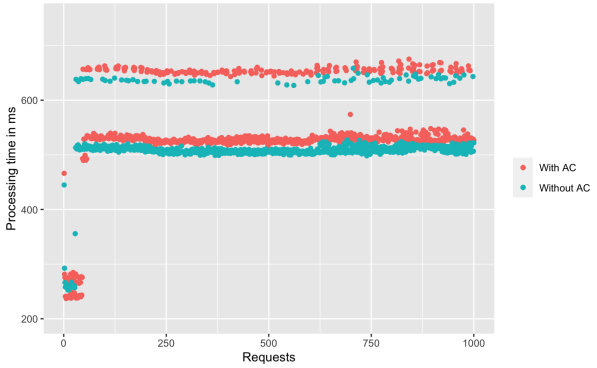


Fig. 6. Second Test (Germany)

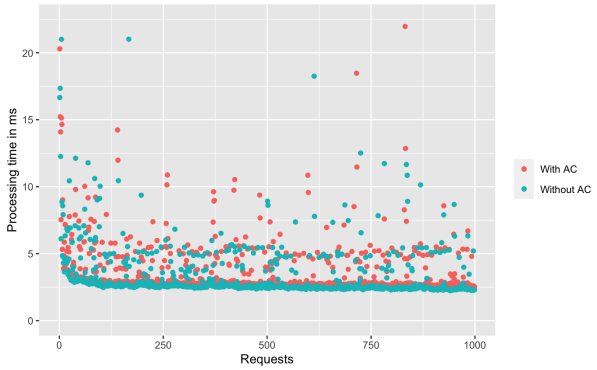
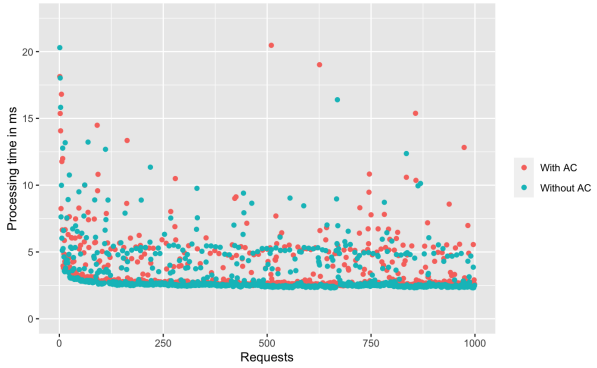


Fig. 7. Second Test (Central US)



In conclusion, the results between the German and US VM is quite big. We expected the same result in both cases, but the results for the central US environment had an average overhead of 5,91 percent while the average overhead for the German environment had been 10,13 percent. Based on these results it is plausible that the access control plugin creates an overhead of around 5 to 10 percent, not considering the additional overhead when logging the processing time.

VII. Challenges

Challenges that can be addressed in the future is to improve the benchmark. To perform our benchmark we used a logging plugin on top of the Apollo server. The overhead created by the logger with this is not deducted from our results. Therefore, in reality the performance could be better by an unknown factor. A more sophisticated benchmark could get more accurate results.

The introduced access control plugin only modifies the results of a processed request. The query for the database stays the same, although some fields are hidden. A more sophisticated way, which could also increase performance, would be to modify the request before it gets resolved by modifying the queries for the database.

VIII. Conclusion

Privacy is an important factor in today's electronic world as information systems increase in size with unprecedented speed. Information can be easily stored, captured and shared in magnitudes and ways that were never seen before. In this kind of environment privacy engineering becomes more important. Access control can ensure privacy protection and help service provider to respect personal data. Privacy tools and their implementation are often dependent on the appropriateness as well as the costs that comes with it. This paper presents an easy to implement rule and purpose based access control plugin for Apollo servers. Based on this report, service providers can now evaluate our findings and consider our plugin to implement access control into their systems. The plugin is open-source and available on GitHub [20].

References

[1] J. Whitman, "The two western cultures of privacy: Dignity versus liberty," *The Yale Law Journal*, vol. 113, 12 2003.

[2] B. News, "Facebook to pay record $5bn to settle privacy concerns," 2021.

[3] S. Gurses and J. Del Alamo, "Privacy engineering: Shaping an emerging field of research and practice," *IEEE Security Privacy*, vol. 14, pp. 40–46, 03 2016.

[4] L. Olejnik, S. Englehardt, and A. Narayanan, "Battery status not included: Assessing privacy in web standards," *CEUR Workshop Proceedings*, vol. 1873, pp. 17–24, 2017. Funding Information: ACKNOWLEDGMENTS We would like to thank Hadley Beeman (W3C TAG), Mar-cos Caceres (Mozilla) and Anssi Konstiainen (Intel) for help and useful feedback. Englehardt and Narayanan are supported by NSF award CNS 1526353. Measurements were funded with an AWS Cloud Credits for Research grant.; 3rd International Workshop on Privacy Engineering, IWPE 2017 ; Conference date: 25-05-2017.

[5] H. Mouratidis, N. Argyropoulos, and S. Shei, *Security Requirements Engineering for Cloud Computing: The Secure Tropos Approach*, pp. 357–380. 07 2016.

Fig. 8. Third Test (Germany)

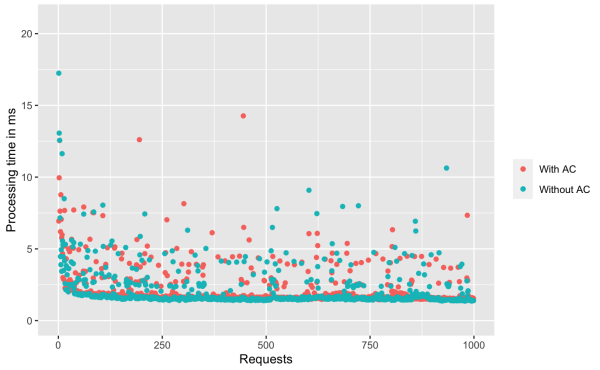
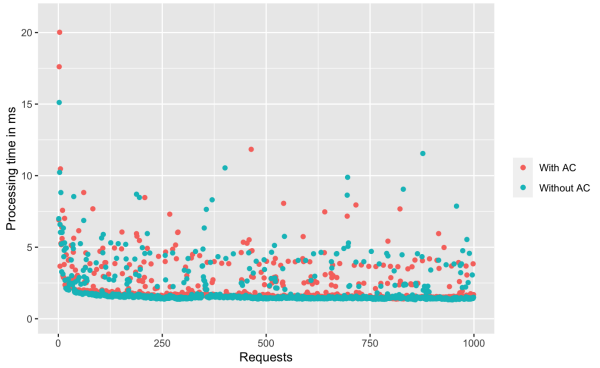


Fig. 9. Third Test (Central US)



[6] H. S. Fhom and K. M. Bayarou, "Towards a holistic privacy engineering approach for smart grid systems," in *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 234–241, 2011.

[7] L. Cranor and N. Sadeh, "A shortage of privacy engineers," *Security Privacy, IEEE*, vol. 11, pp. 77–79, 03 2013.

[8] M. Abadi, "Logic in access control," in *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.*, pp. 228–233, 2003.

[9] H. Shen and P. Dewan, "Access control for collaborative environments," in *Proceedings of the 1992 ACM Conference on Computer-Supported Cooperative Work*, CSCW '92, (New York, NY, USA), p. 51–58, Association for Computing Machinery, 1992.

[10] N. Kashmar, M. Adda, and M. Atieh, *From Access Control Models to Access Control Metamodels: A Survey*, pp. 892–911. 01 2020.

[11] B. Beilman, "Rule-based vs. role-based access control."

[12] J.-W. Byun and N. Li, "Purpose based access control for privacy protection in relational database systems," *VLDB J.*, vol. 17, pp. 603–619, 07 2008.

[13] G. Roden, "Was man über GraphQL wissen sollte."

[14] "Queries and Mutations | GraphQL."

[15] "GraphQL erklärt."

[16] M. Zavadlal, "maticzav/graphql-shield," May 2021. original-date: 2018-02-11T17:22:18Z.

[17] T. Hata, "hata6502/graphql-filter," Jan. 2021. original-date: 2020-08-06T13:58:49Z.

[18] F. Pallas, M.-R. Ulbricht, S. Tai, T. Peikert, M. Reppenhagen, D. Wenzel, P. Wille, and K. Wolf, "Towards application-layer purpose-based access control," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, (New York, NY, USA), p. 1288–1296, Association for Computing Machinery, 2020.

[19] A. G. Docs, "Introduction to apollo server," 2021. original-date: 2021-05-26T23:02:21Z.

[20] H. V. Nguyen and W. Klausing, "wklausing/apolloacplugin."