Hook-In Privacy Capabilities for gRPC

Jiaao Li, Riccardo Marin, Timo Ramsdorf

Privacy Engineering, TU Berlin

14th July 2022

Overview

Recap

Goal:
Create a component which introduces privacy functionalities into a gRPC-based use case.

↓

Related work :

- *"Towards Application-Layer Purpose-Based Access Control"*
- *"Configurable Per-Query Data Minimization for Privacy-Copliant Web APIs"*

  ...

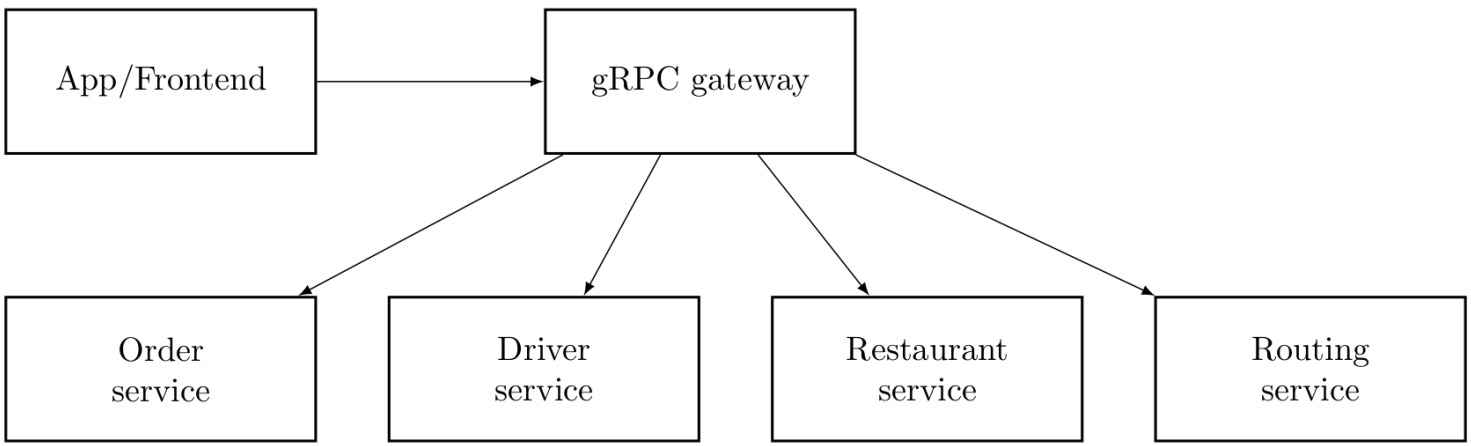
↓

Capabilities of component:

- Approach: Interceptors
- Data Minimization: Erasure, Generalization, Noising, Hashing
- Purpose Based Access Control

Recap: Use Case

- gRPC - a high performance Remote Procedure Call (RPC) framework that can connect polyglot services in microservice style architecture
- Use case - a gRPC Based Food Delivery Application



Recap: Use Case

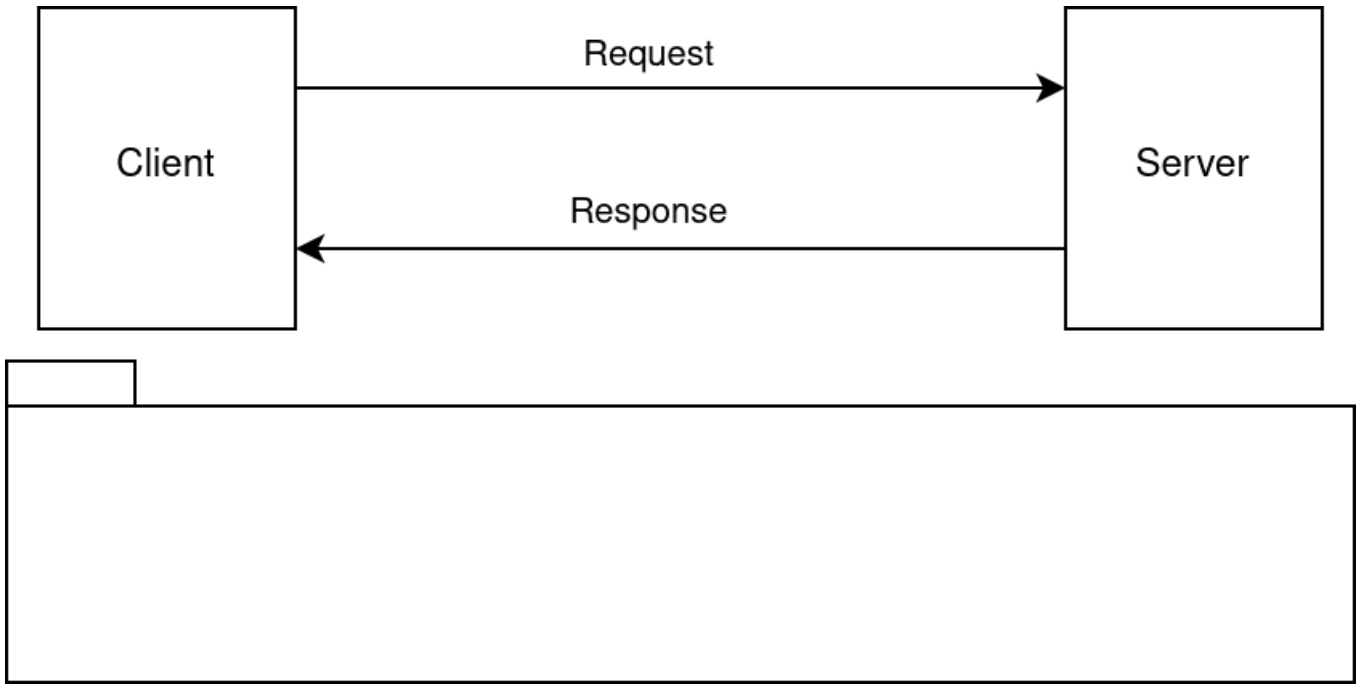
Our Interceptors should minimize unnecessary data and control access for various purposes.

Service	**Data**	**Minimization**
Driver service	Meal	*Erasure*
	Customer info	*Erasure*
	Delivery address	
Restaurant service	Customer info	*Erasure*
	Delivery address	*Generalization*
	Driver identity	*Hashing*
	Meal	
Routing service	Driver location	*Noising*

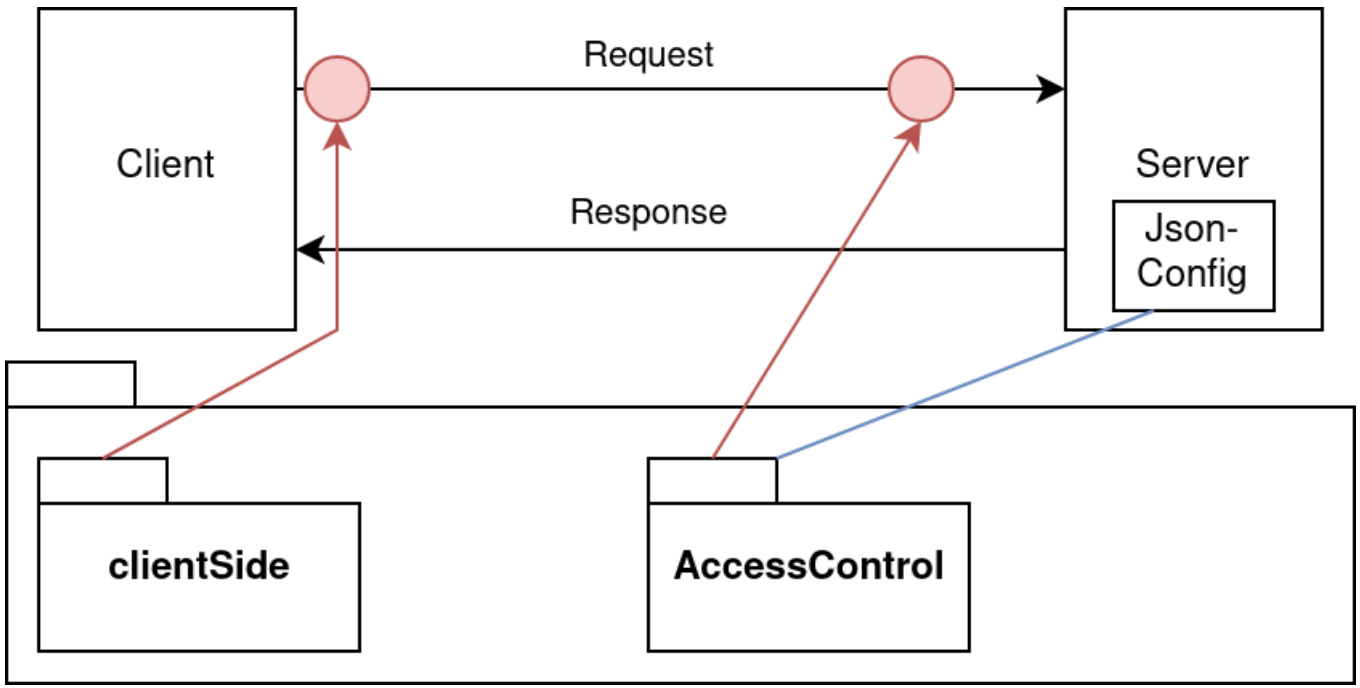
Design Principles

1. HOOK-IN
   Builder-Pattern, Json-Config, Default over Configuration
2. PERFORMANCE
   put as much work as possible in the initialization
3. MODULARITY
   clientSide, Authorization, accessControl, dataMinimization
4. EXTENDABILITY
   API for custom minimization functions

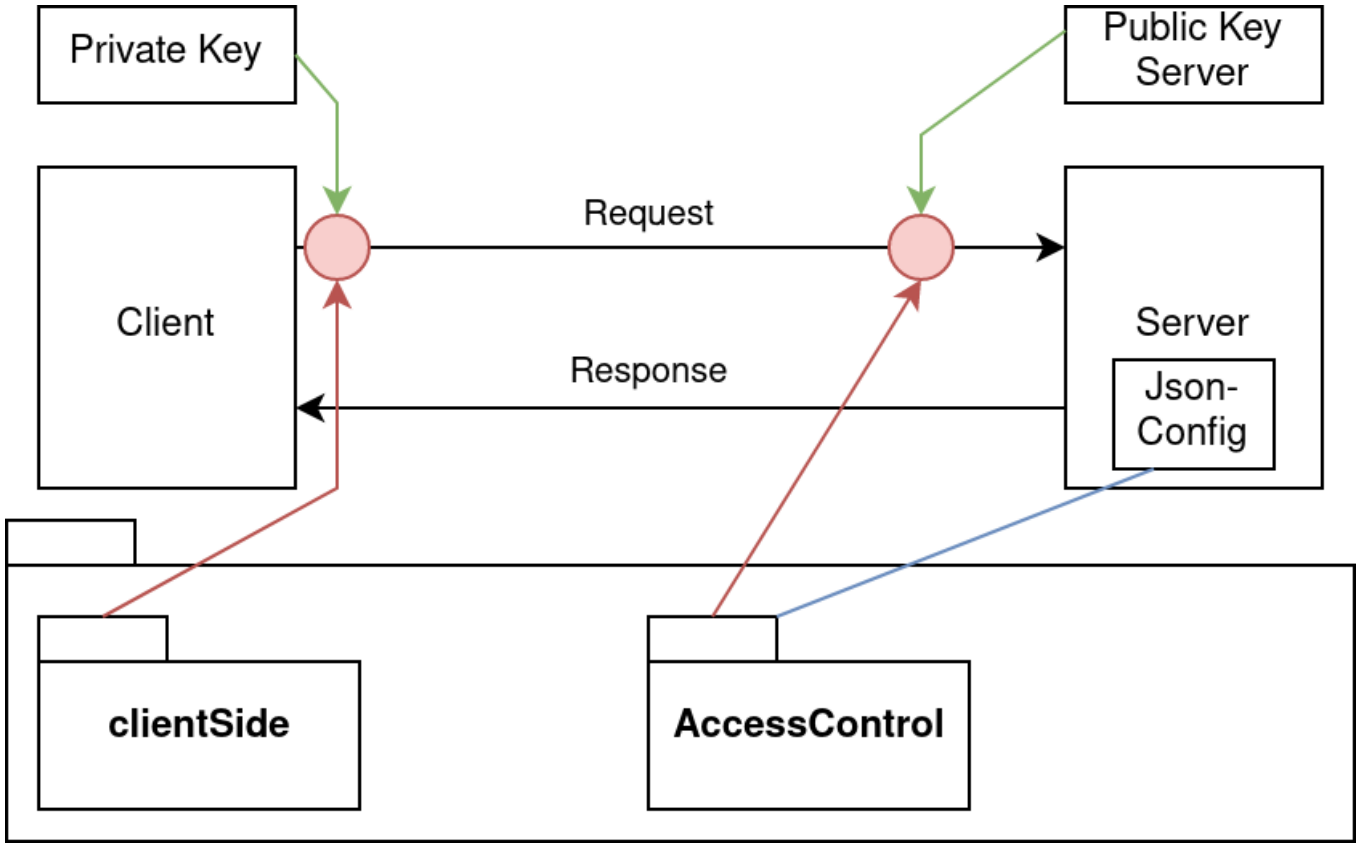
Component Architecture



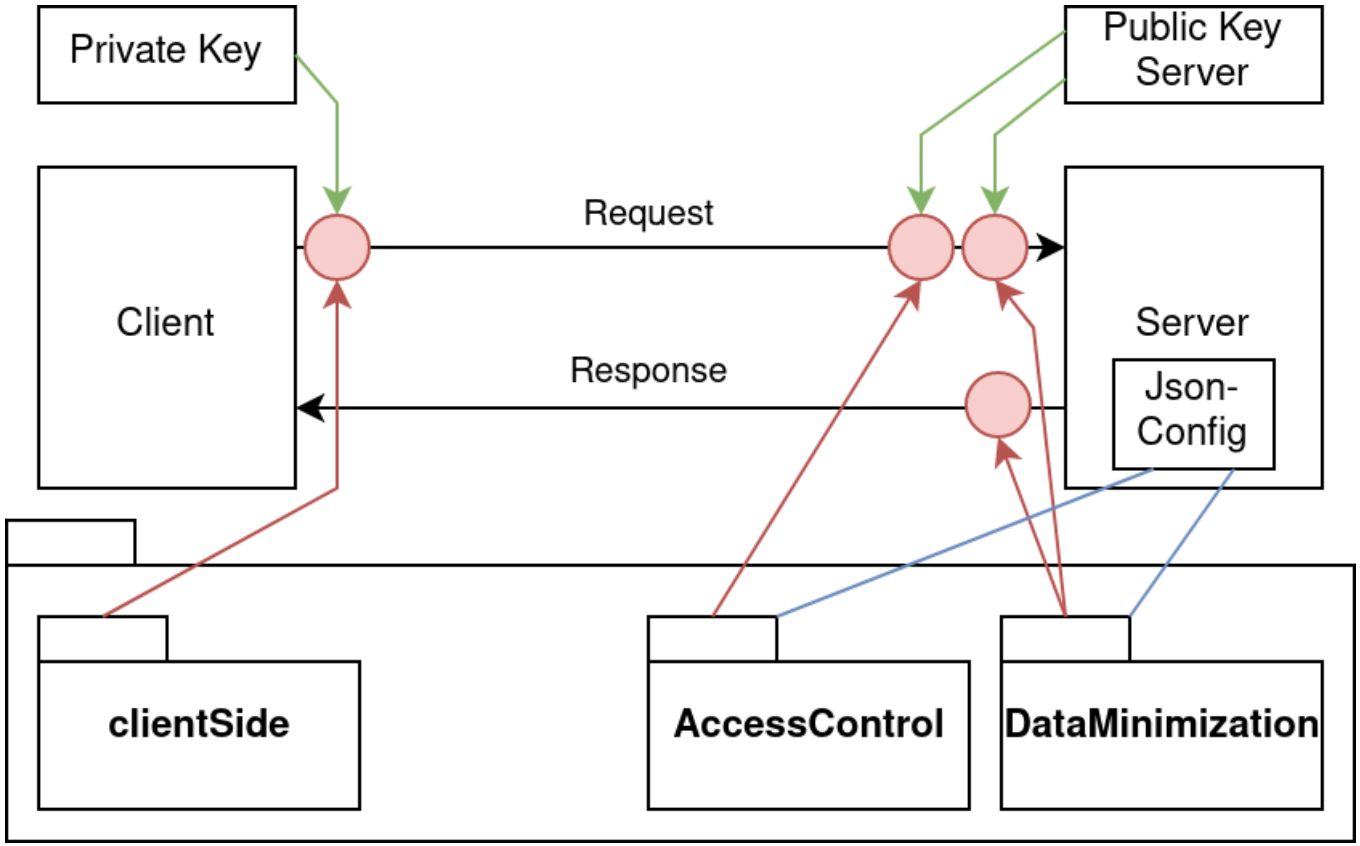
Component Architecture



Component Architecture



Component Architecture



Implementation: Data Minimization

Standard server creation in GRPC:

```
server = ServerBuilder.forPort(this.port) ServerBuilder<capture of ?>
        .addService(new OrderImpl()) capture of ?
        .build() Server
        .start();
```

Implementation: Data Minimization

Easiest use case for data minimization:

```
String configPath = Paths.get( first: ".").toAbsolutePath().normalize() + "/config.json";
server = ServerBuilder.forPort(this.port) ServerBuilder<capture of ?>
        .addService(new OrderImpl()) capture of ?
        .intercept(DataMinimizerInterceptor.newBuilder(configPath).build())
        .build() Server
        .start();
```

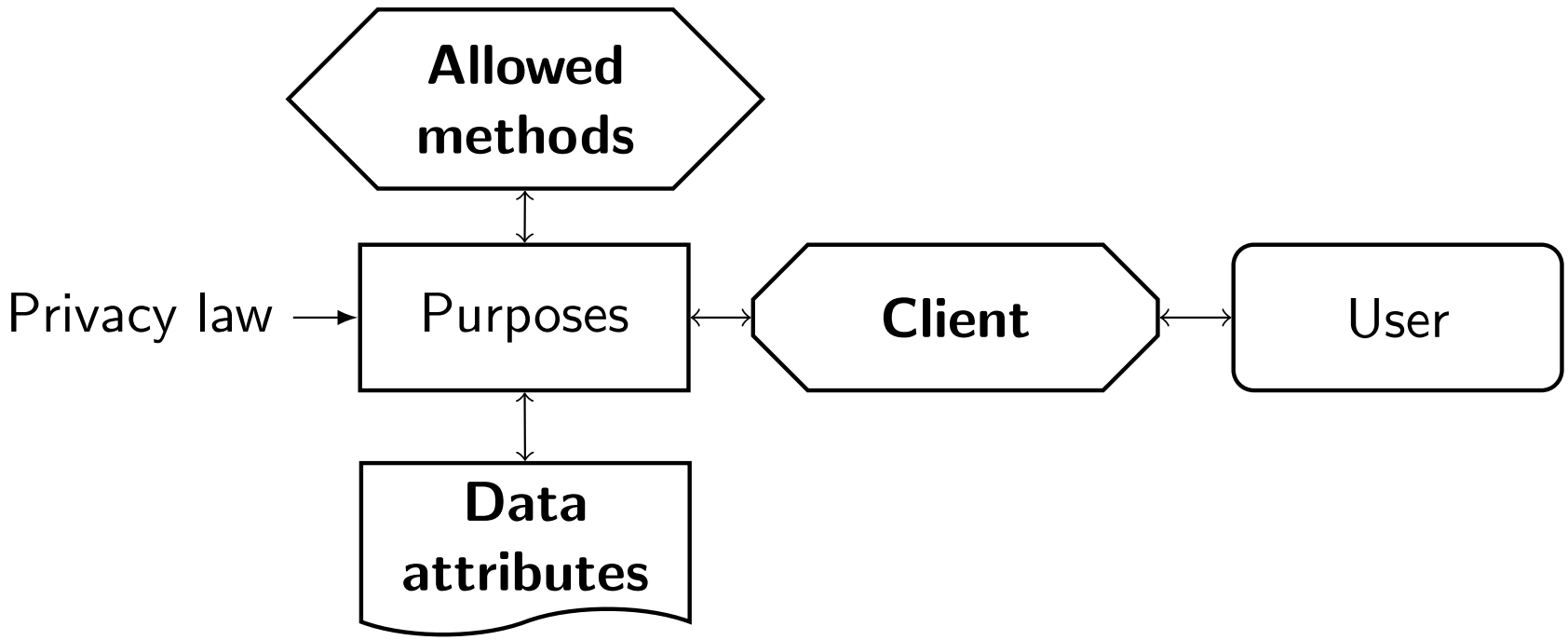
Implementation: Data Minimization

Data Minimization with custom function:

```
String configPath = Paths.get( first: ".").toAbsolutePath().normalize() + "/config.json";
MinimizationFunction mf = new MinimizationFunction()
        .addStringOperator((value, config) -> {
            String readableEnd = config.getOrDefault( key: "readableEnd",  defaultValue: "0");
            int hiddenPrefix = value.length() - Integer.parseInt(readableEnd);
            return "*".repeat(hiddenPrefix) + value.substring(hiddenPrefix);
        });
server = ServerBuilder.forPort(this.port) ServerBuilder<capture of ?>
        .addService(new OrderImpl()) capture of ?
        .intercept(DataMinimizerInterceptor.newBuilder(configPath)
                .withoutRequestIntercepting()
                .defineMinimizationFunction( name: "starReplace", mf)
                .build())
        .build() Server
        .start();
```

Implementation: Access Control

Configuration of the Purpose-based Access Control



Implementation: Access Control

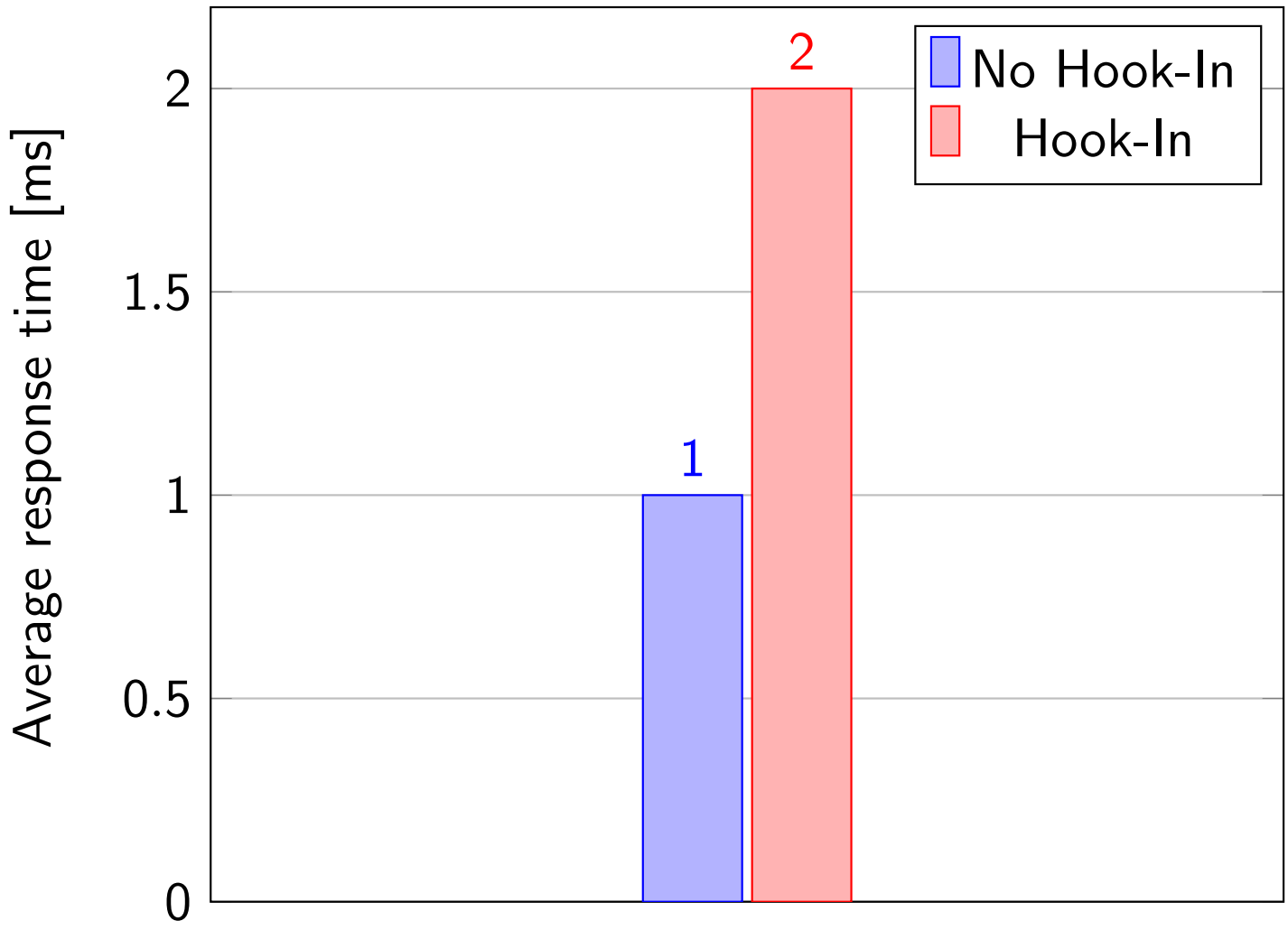
Access Control is based on the content of the request header:

1. Key server exposes public keys upon request
2. Client attaches to the request header a JWT token signed with its private key, the client name and purpose of the request.
3. The server checks the signature using the public key and whether the client-purpose-method combination is allowed

Implementation: Configuration

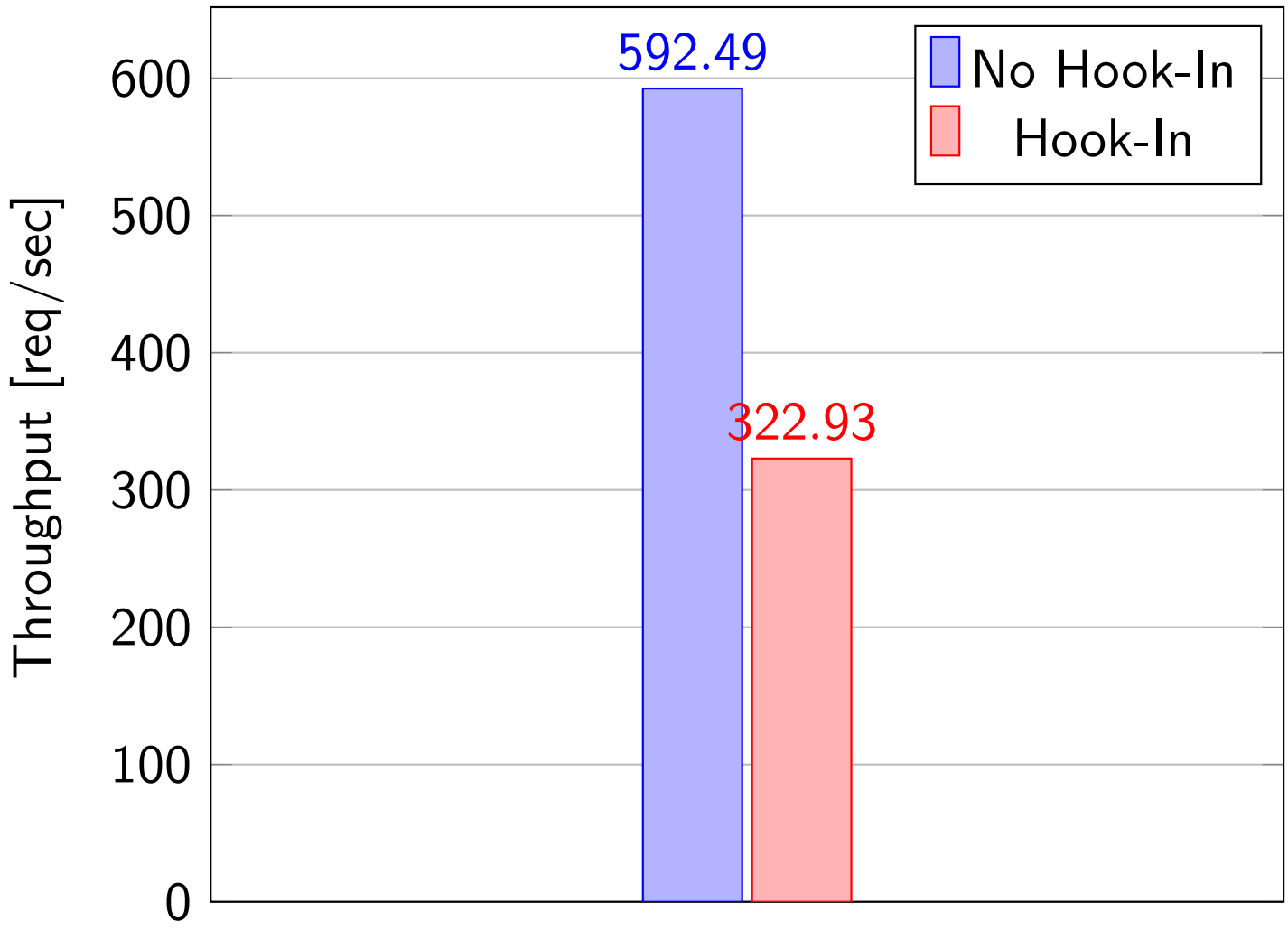
```
{
  "key_server": {
    "host": "localhost",
    "port": 50005
  },
  "purposes": {
    "meal_purchase": {
      "name": "Meal purchase",
      "allowed_clients": [ "client", "order" ],
      "allowed_methods": [ "proto.OrderService/OrderMeal" ],
      "minimization": {
        "OrderRequest": {
          "name": [
            {
              "function": "replace",
              "replace": "Mister X"
            }
          ],
          "surname": [
            {
              "function": "erasure"
            }
          ]
        }
      }
    },
```

Benchmark



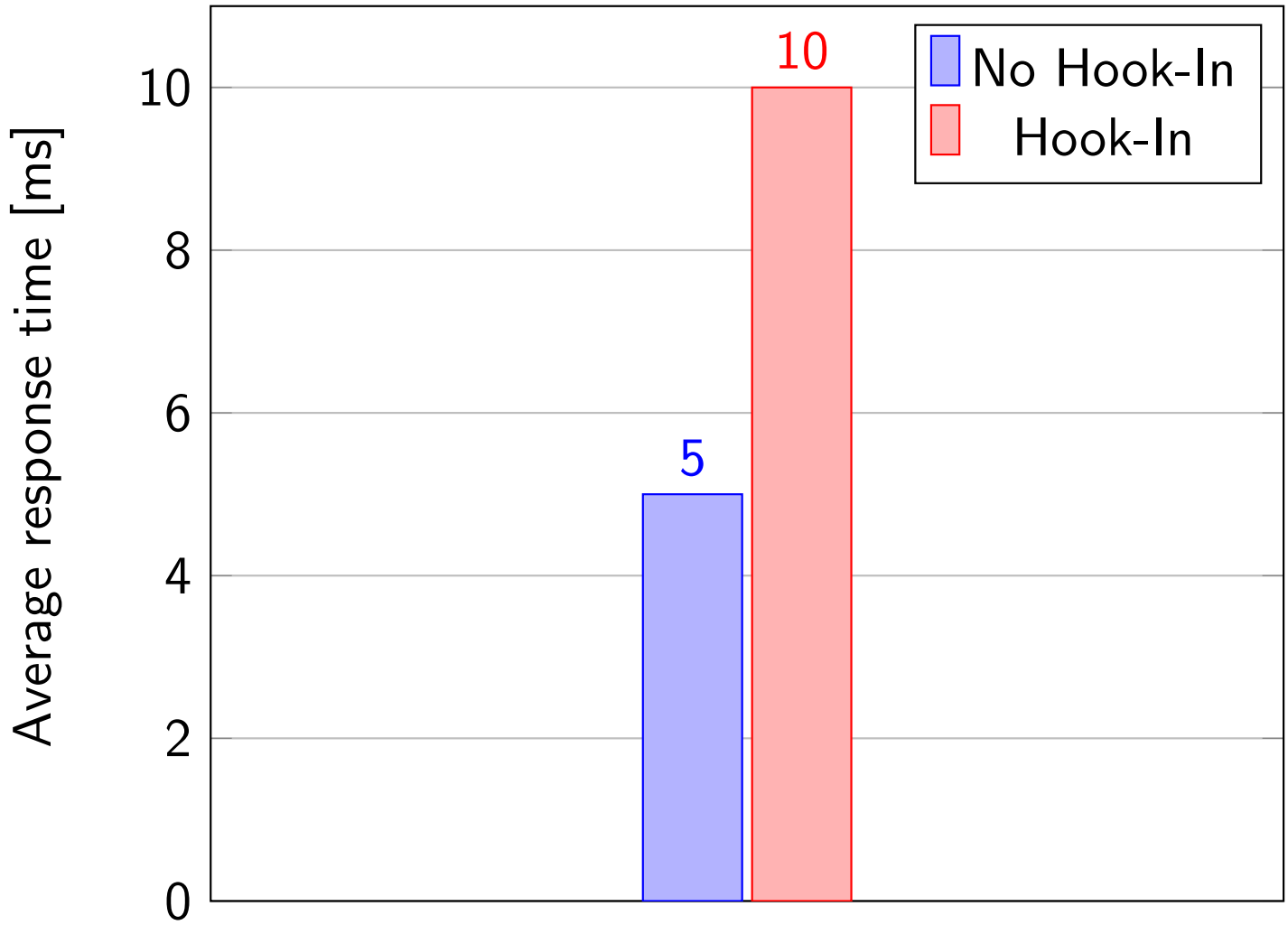
5000 requests, simple request and response between 2 gRPC clients

Benchmark



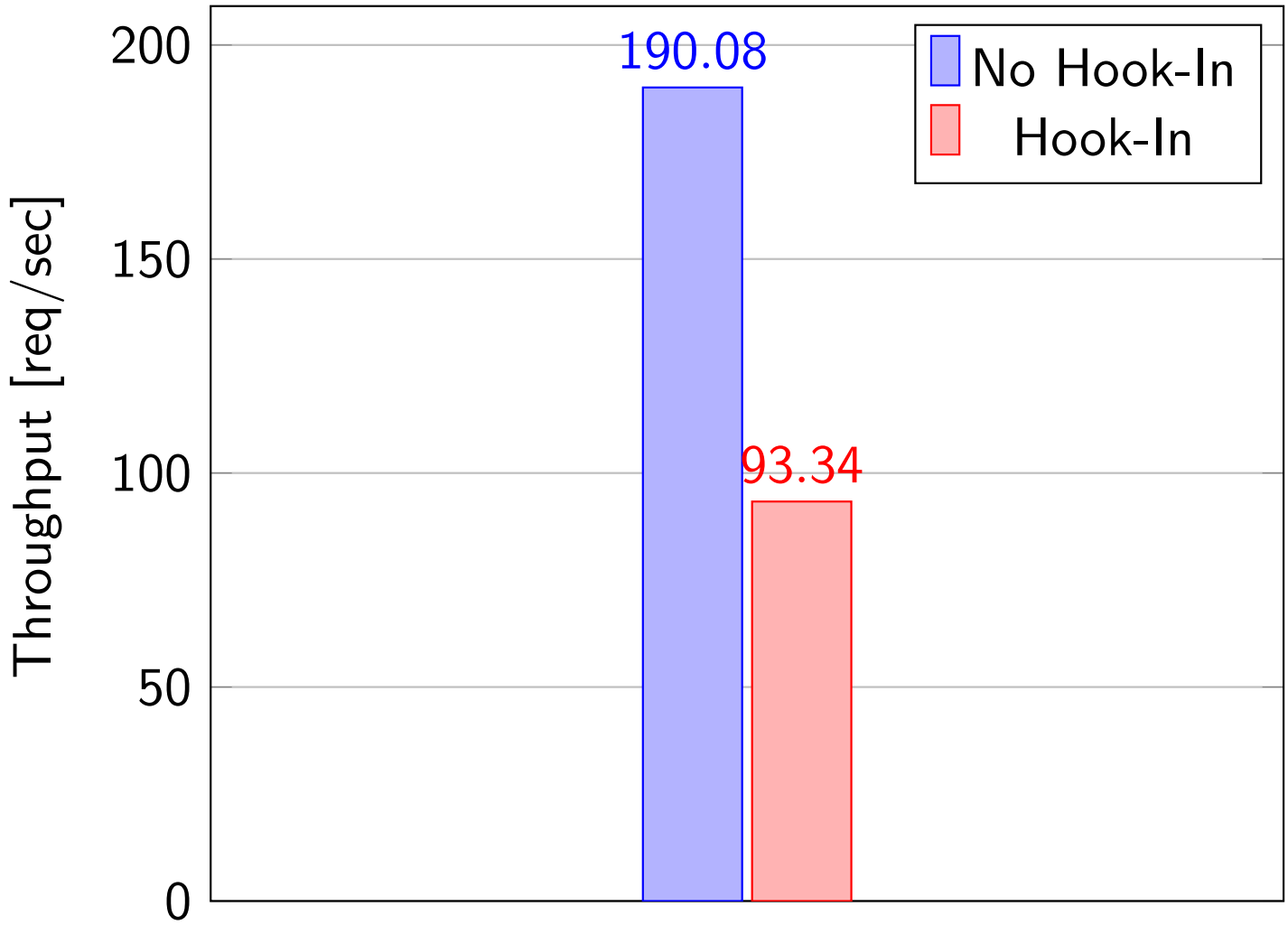
5000 requests, simple request and response between 2 gRPC clients

Benchmark



5000 requests, simulation of the entire use case (5 different gRPC entities exchanging data)

Benchmark



5000 requests, simulation of the entire use case (5 different gRPC entities exchanging data)

Future work

Based on the benchmarking results, the future work should focus on the performance optimization.
Possible ideas:

- Find a more efficient solution for key exchange and signature verification
- Go-based interceptors may have better performance
- Another potential solution: Binary Proxy

Conclusions

We proved that it is possible to build a hook-in privacy component for gRPC that...

- is usable for existing gRPC application
- provides already some data minimization techniques
- allows custom minimization function
- provides client authorization internally

The component may fit some use cases where performance is not crucial.

Demo

Live demo

Q&A

Q&A Time