Large scale performance test for security policies

Shared with Istio Community



Owner: Michael Eizaguirre **Contributors:** Yangmin Zhu, Carolyn Hu **Working Group:** Istio Security	**Status**: WIP | **In Review** | Approved | Obsolete **Created**: 2020-08-05 **Approvers**: N/A.

Overview

Today we don't have data about how does Istio AuthN/Z policy behave in large scale environments, for example the policy distribution latency, runtime overhead, and the resource usage (CPU/memory) when there are 10k policies or a single policy with 10k rules.

We want to be able to have a better understanding of what kinds of performance changes occur when a user has hundreds of AuthZ/AuthN policies or few policies with many rules.
These performance tests will be able to give our customers a higher confidence in adopting security policies and also help us discover bottlenecks.

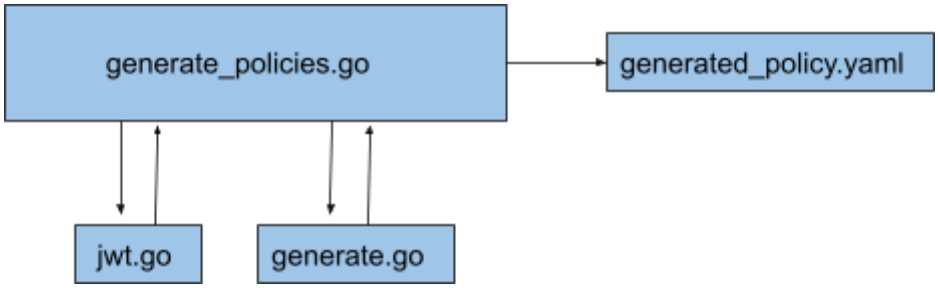
Goals

Understand the latency/CPU/mem usage relationship with different kinds of security policy use cases.

Design Overview

To be able to run these performance tests the first thing we need to do is be able to generate AuthZ/AuthN policies with a high degree of flexibility (Flexibility as in the number of policies, the kind of policies, the number of rules etc). To be able to create a large number of policies we will use a go file named generate_policies.go which was created for this particular purpose. The overall structure of generate_policies is shown below.

Shared with Istio Community



Summary

Use Guide

For the user to run these specific performance tests the flow will be very similar to the existing flow for perf benchmarks. The difference is that the user will need to generate the wanted policy, apply the policy, and then run the perf tests, this will be optimized by creating config files. Currently working on creating some script files which will run the major test cases.

User experience:

Currently the user needs to follow these steps to run the performance tests: (Steps 1, 2, and 4 are explicitly explained in the README for generate_policies)

Step 1: Generate the wanted policy using generate_policies.go
Step 2: Apply the policy
Step 3: Run existing perf tests
Step 4: Cleanup applied policies

The next step to improve the multistep user experience will be achieved by creating a premade set of useful test cases and placing them in config files. These config files will generate the policies, apply them, run the perf test, and then cleanup any applied policies all with a single command. The details of the specific use cases will be explained more in the design portion of the doc.

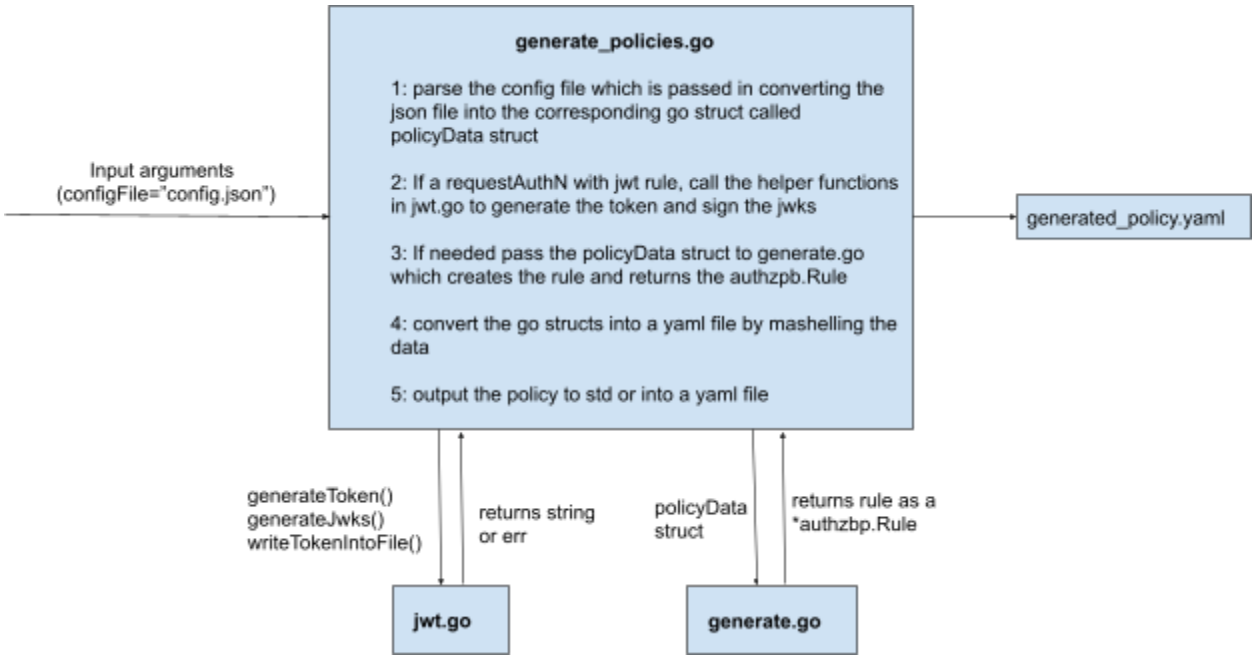
Design

Shared with Istio Community

Currently we are mainly testing DENY policies (testing ALLOW for jwks rules), the policies being created are not meant to match, this will make sure no rule will allow a short circuit by matching early and not going through all the rules, producing inaccurate data.

Configurable parameters:	actual flag:
• The total number of policies to create	numPolices
• The namespace in which the policy will be applied to	namespace
• The action of the rule (DENY/ALLOW)	action
• Policy Types (PeerAuthentication, RequestAuthentication AuthorizationPolicy)	(see README)
• The number of paths in each rule	numPaths
• The number of sourceIP's	numSourceIP
• The number of JWKS rules	numjwks
• The number of source request principals	numRequestPrincipal
• The number of source principals	numPrincipal
• MTLS mode (STRICT, DISABLE, PERMISSIVE)	mtlsMode
• Create an InvalidToken	InvalidToken
• Set the Token Issuer	tokenIssuer

The main interface that is used to create policies is generate_policies.go. What generate_policies does is to parse the input flags, calls generate.go which creates the specific rules for each AuthZ/AuthN policy



With a simple config.json file contains the following

Shared with Istio Community

```
{
  "authZ":
  {
    "numPolicies":1,
    "numSourceIP":1
  }
}
```

The corresponding output yaml will look as follows

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: test-1
  namespace: twopods-istio
spec:
 action: DENY
 rules:
 - from:
   - source:
       ipBlocks:
       - 0.0.0.0
```

The existing performance tests are quite thorough and allow users to easily see the latency of requests, and by using prometheus we can get the cpu/mem usage of the istio proxy. This allows us to not need to implement any added functionality to get the data we are trying to obtain.

Alternative design:

One design alternative was more a different design in the integration with runner.py
A route we explored was to be able to add functionality to runner.py so that if passed in a security test flag it would call generate_policies, apply the policy, and then run the normal perf test. An issue with this approach is that it adds functionality to runner.py that is not meant to be done in runner.py. Even though it would have made the experience easier for a user we determined it would not be good practice to add functionality to runner.py that would be very specific for the security performance tests.

Pros: Simpler use case for the user. Instead of needing to manually create the test

Shared with Istio Community

Cons: Adds functionality to runner.py which is not something that is intended for runner.py to do.

Implementation locations:

generate_policies/

Most of the changes occurring in the creation of generate_policies folder which contains generate_policies.go, generate.go, jwt.go, and the yaml files that are created by generate_policies.go.

runner.py

Minor changes will need to be done in runner.py in relation to adding the security policy Flag. This flag will allow for easy queries when searching through the output csv that is Produced by fortio

jwt.go

Used to generate tokens as well as sign and produce jwks for requestAuthentication rules.

perf/benchmark/config or generate_policies/config

To allow users to easily run performance tests there needs to be some config files. They can either be stored in the perf/benchmark/config folder or in a separate folder Inside of generate_policies which contains the config files and the corresponding json Files needed.

Fortio

Other changes which will need to occur are inside of fortio. As fortio currently does not have an extra label field within it noting that a security test was run, we are in the talks with the CSM telemetry team to create an extra label field within the output of fortio which is dynamically created and would contain the security label.

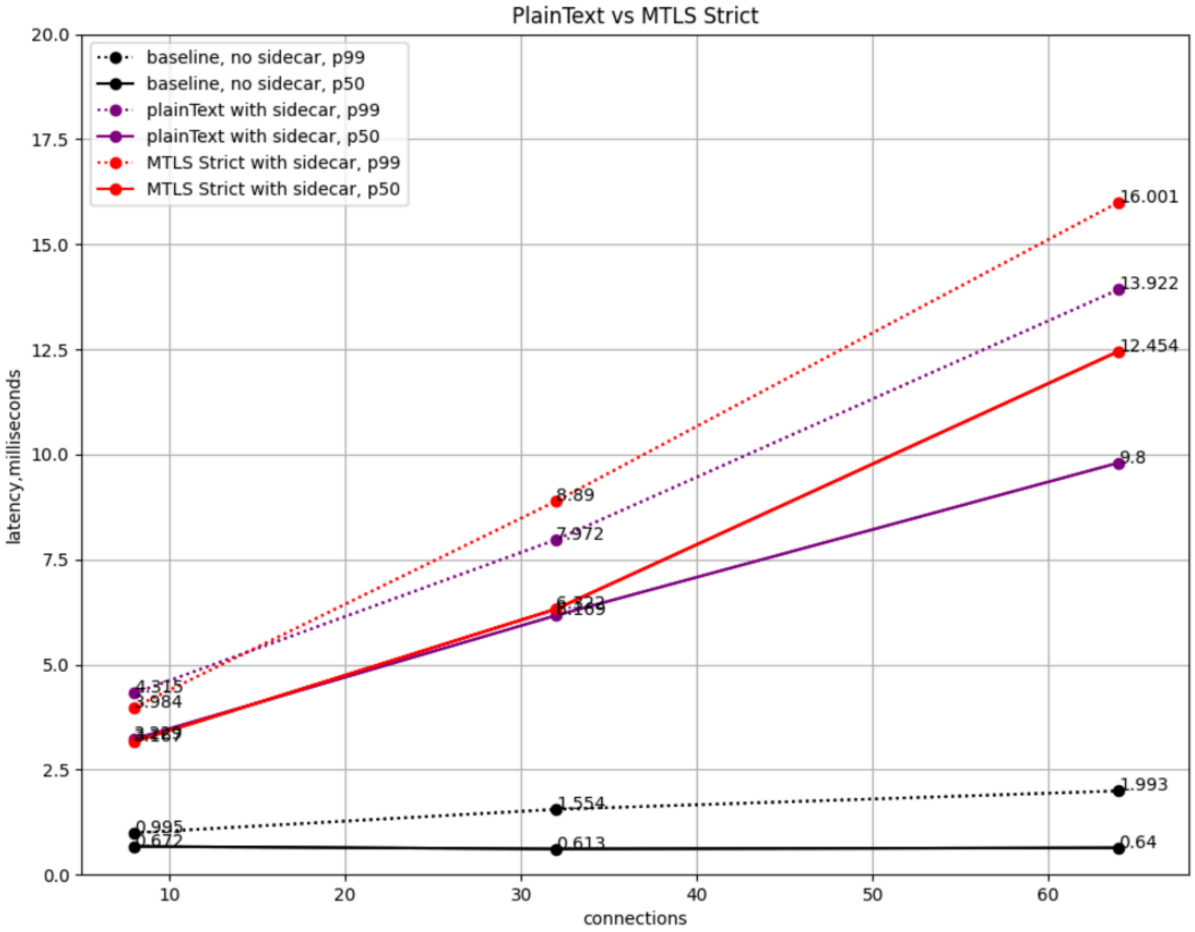
Data so far:

Since there are many variables that can be adjusted for the sake of simplicity the following graphs will all have the same x and y axis, the y access will have the latency in milliseconds, and the x access will the number of connections used for a test. This may be slightly misleading but the x access runs in a small, medium, large amount of connections as well as qps.

1: connections=8, qps=100 (small)
2: connections=32, qps=500 (medium)
3: connections=64, qps=1000 (large)

Shared with Istio Community

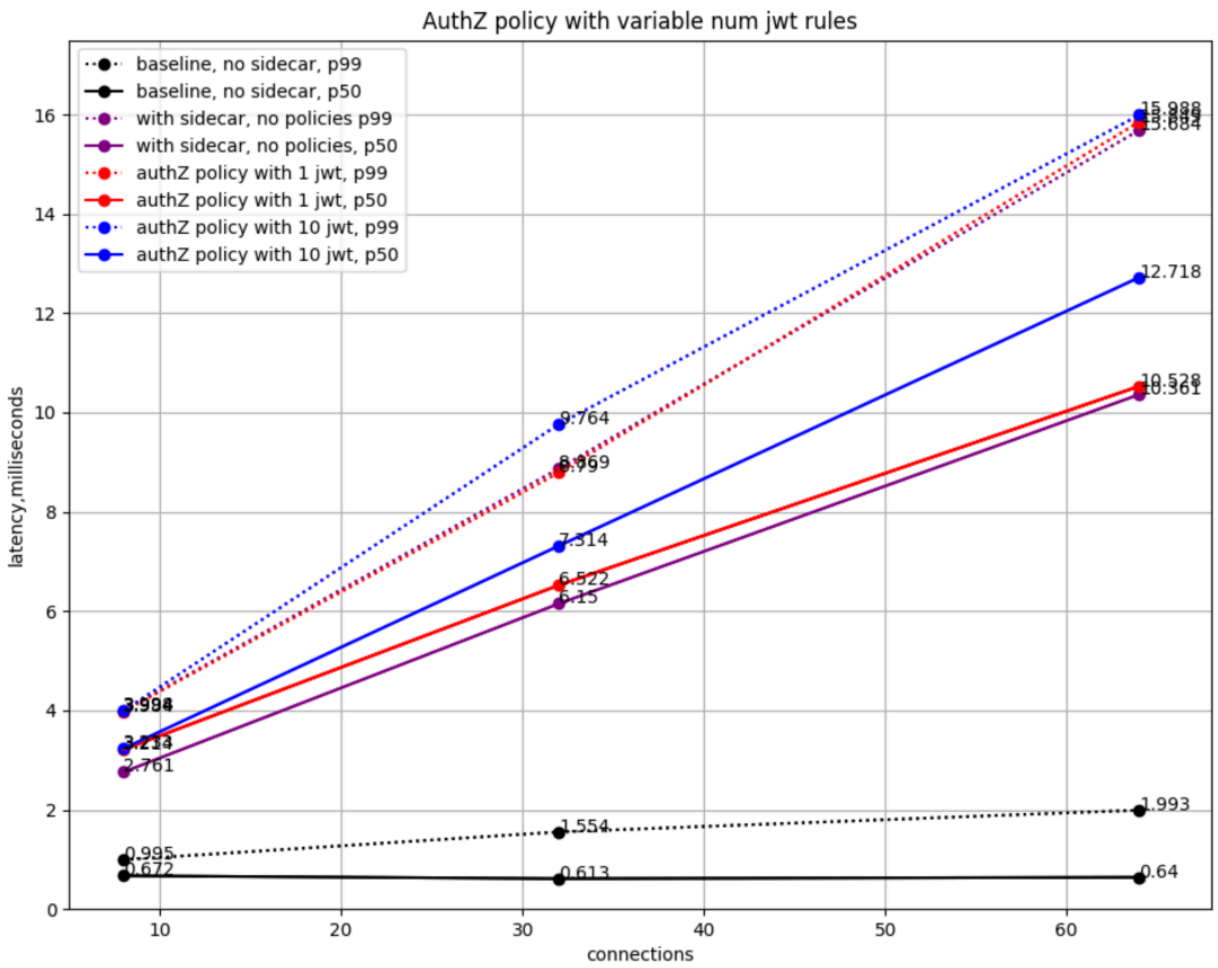
Test case 1: (PeerAuthentication - Test plaintext and mTLS)



Source of data

Test case 2: (RequestAuthenticaion - Test different number of jwks)

Shared with Istio Community



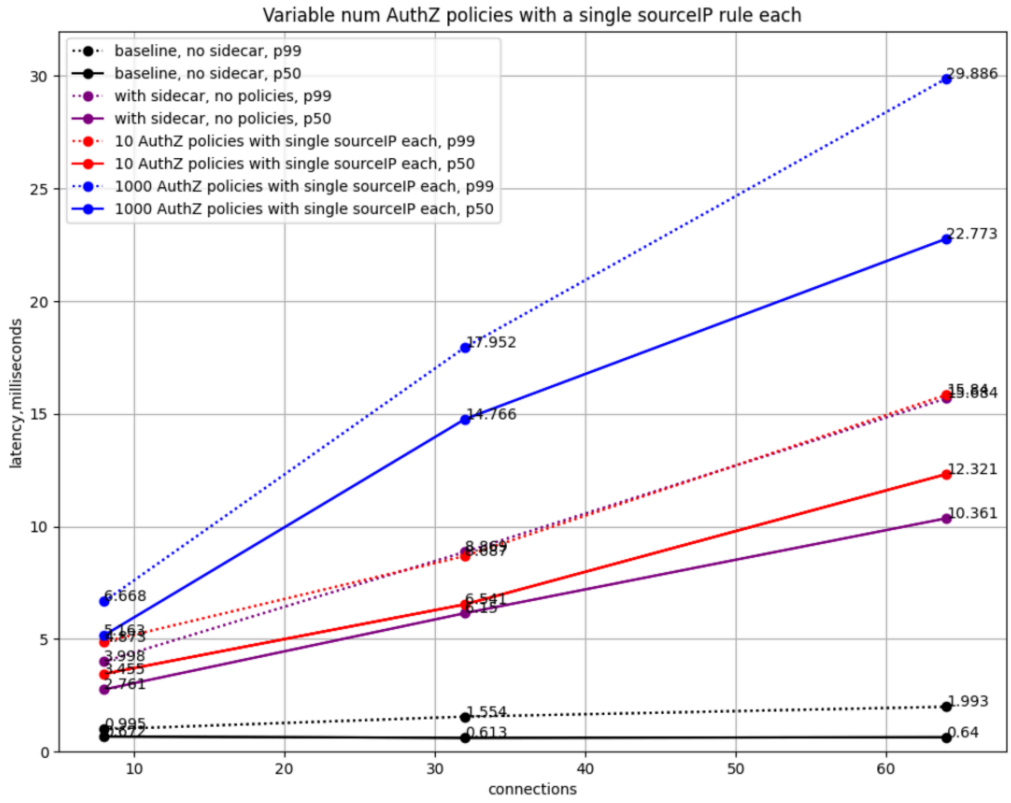
Source of [data]

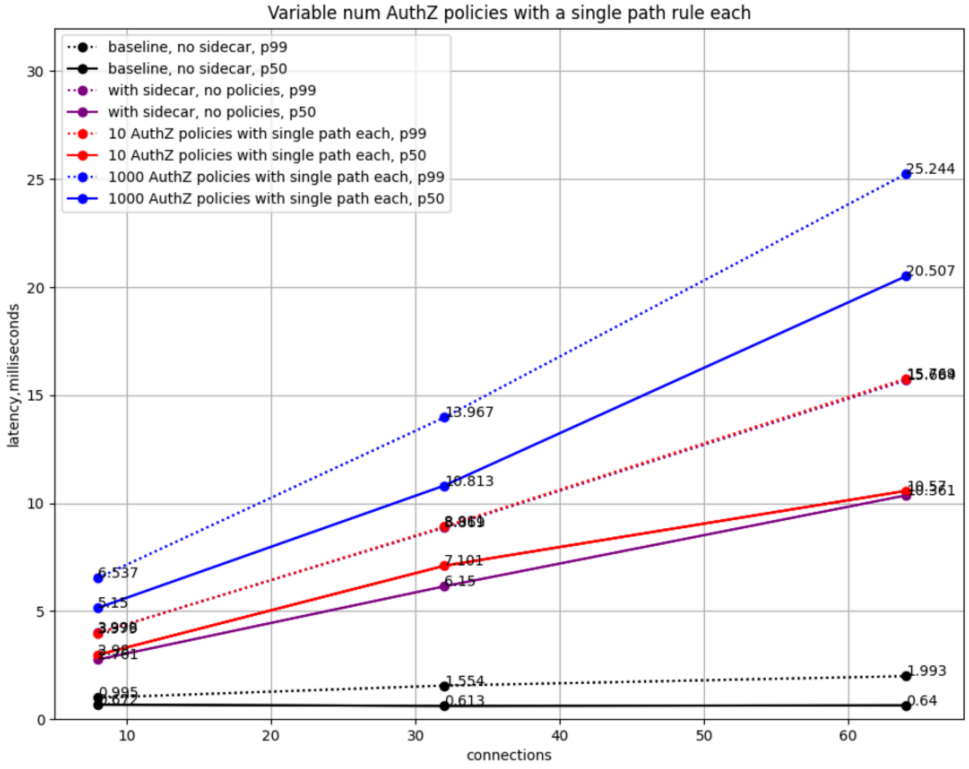
As expected with a larger number of jwks rules the higher the latency. Here the baseline is in black the baseline of having the sidecars deployed but with no policies is purple, authorization policy with 1 jwks is in red and the authorization policy with 10 jwks is in blue. This test can also be automatically run by running the command inside of generate_policies/testRequestAuthN (currently not in istio/tools)

Test case 3: **(AuthorizationPolicy - Test path and sourceIP)**

Case 1: 0 (baseline) vs 1 policy vs 1000 policies. This test consists of creating a variable number of AuthZ Policy which has 1 rule which we will compare the rule being path or from sourceIP

Shared with Istio Community

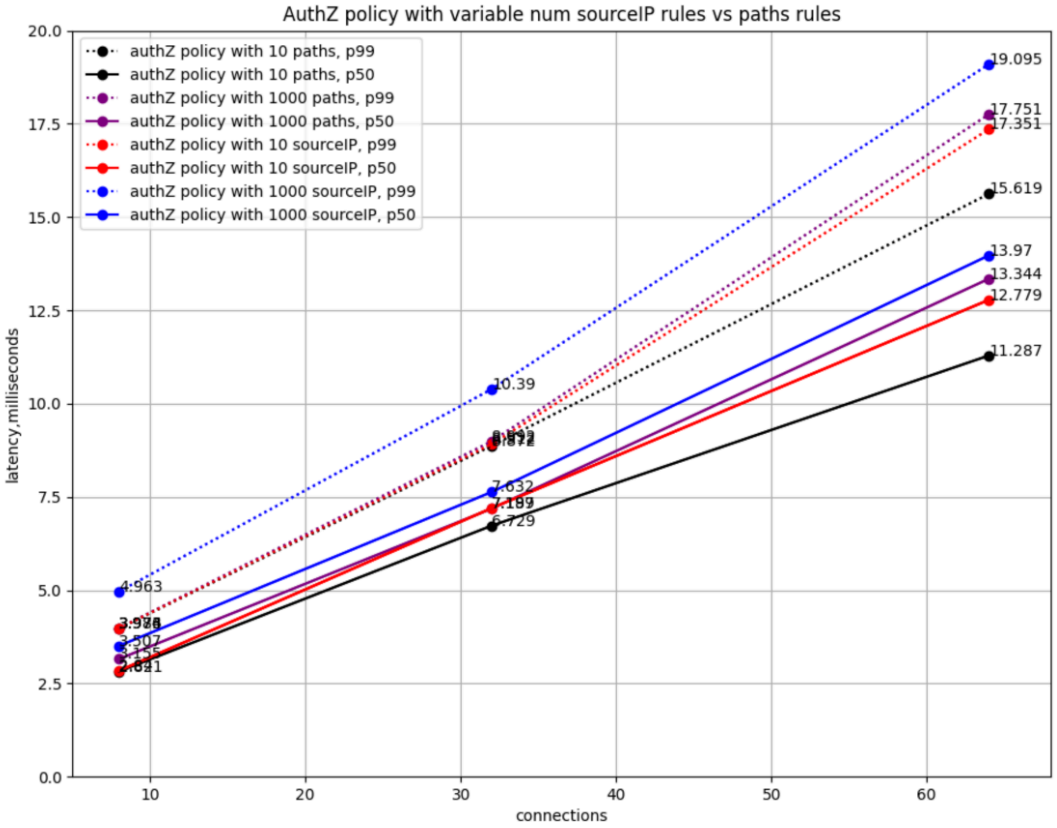




Source of [data]

Shared with Istio Community

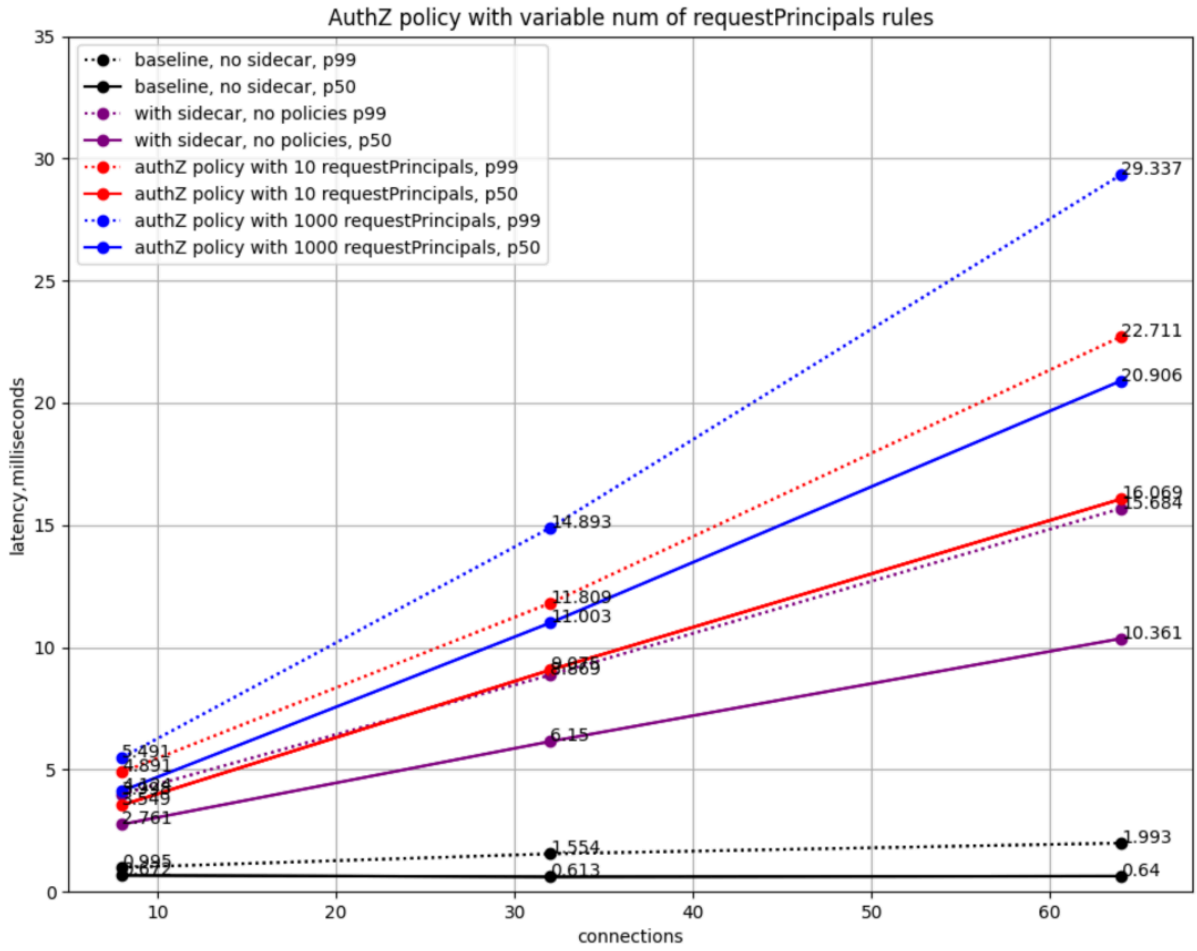
Case 2: 0 (baseline) vs 1 rule vs 1000 rules. This test consists of create a single AuthZ Policy which has a variable number of paths vs sourceIP's



Source of [data]

Test case 4: (RequestAuthentication + AuthorizationPolicy)

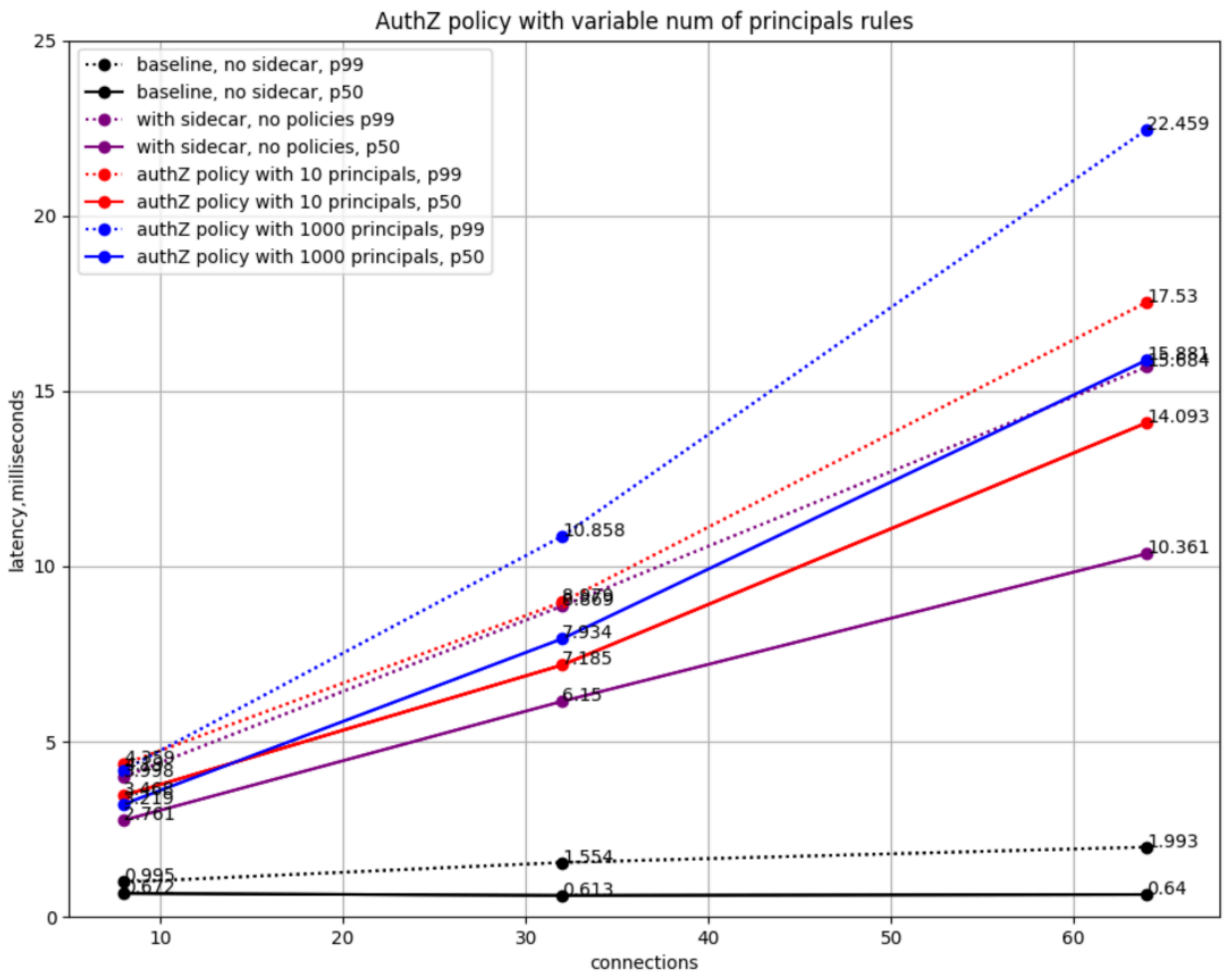
Case 1: 0 (baseline) vs 1 vs 1000 requestPrincipals. This test consists of create a single requestAuthN rule and a single authZ rule which contains either 1 or 1000 requestPrincipals



Source of data

Test case 5: (PeerAuthentication + AuthorizationPolicy)

Case 1: 0 (baseline) vs 1 vs 1000 Principals. This test consists of create a single peerAuthN rule and a single authZ rule which contains either 1 or 1000 source Principals



Source of [data]

Shared with Istio Community