Solution Brief

Service Mesh – Istio and Envoy Optimizations for Intel® Xeon® Scalable Processors

A service mesh is an infrastructure network layer framework that handles security, traffic management, and telemetry between microservices in a clustered environment.

Authors

Luyao Zhong

Mrittika Ganguli

Ismo Puustinen

Qiming Liu

Rafal Sznejder

Sakari Poussa

Ramesh Masavarapu

Executive Summary

A service mesh is a configurable, low-latency infrastructure layer designed to handle a high volume of network-based inter-process communication among application infrastructure services using application programming interfaces (APIs). The service mesh ensures that the communication layer between microservices is fast, reliable, and secure. Some of the key features include service discovery, security, traceability, and observability.

A service mesh can be implemented by multiple open-source software solutions. This document discusses the open-source project, Istio that implements the service mesh architecture. Istio is the control plane and Envoy is used as a sidecar proxy for the data plane.

This document is part of the Network Transformation Experience Kits.

Service Mesh Performance and Latency Challenges

One of the biggest challenges with the current open-source implementations of Istio and Envoy that have been addressed by Intel are associated with performance and latency.

Service mesh deployments that use Istio with Envoy cause latency and performance challenges due to the nature of sidecar implementation in Envoy. Intel has addressed this performance and latency challenges by utilizing Intel® Xeon® CPU features such as Intel® QuickAssist Technology (Intel® QAT) and Intel® Dynamic Load Balancer (Intel® DLB). The results are:

- Up to 1.6x CPU cycles saved, up to 2.37x throughput/RPS improvement and up to 1.95x latency reduction using Intel QAT in a 1C-16C scaling experiment on the 4th Gen Intel® Xeon® Scalable processor
- Using 1x QAT, for 8 core and 16 cores, save 42% and 60% respectively on the 4th Gen Intel Xeon Scalable processor CPU cycles
- Using 2x QAT, for 8 core and 16 cores, save 18% and 49% for 2x Intel QAT on the 4th Gen Intel Xeon Scalable processor CPU cycles
- For the same RPS, latency improves upto 1.96x for mixed message sizes using Intel DLB on the 4th Gen Intel Xeon Scalable processor
- Reduction in latencies on the 4th Gen Intel® Xeon® Scalable processor using the Hyperscan optimizations
- TCP/IP eBPF Bypass performance optimization has shown a reduction of latencies on the 4th Gen Intel Xeon Scalable processor

Service Mesh - Istio and Envoy

A service mesh is a dedicated infrastructure layer for handling service-to-service communication. It is responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud native deployment. In practice, the service mesh is typically implemented as an array of lightweight network proxies that are provisioned alongside deployment code, without the deployment needing to be aware. The core of service mesh is to provide a unified global method to control and measure all request traffic between deployments or services.

Istio is an open-source solution, which implements the service mesh framework that handles traffic management, security, telemetry, and observability within a clustered environment. Istio utilizes Envoy as a sidecar proxy to handle all data plane traffic whereas, Istio handles all the control plane traffic within a clustered environment.

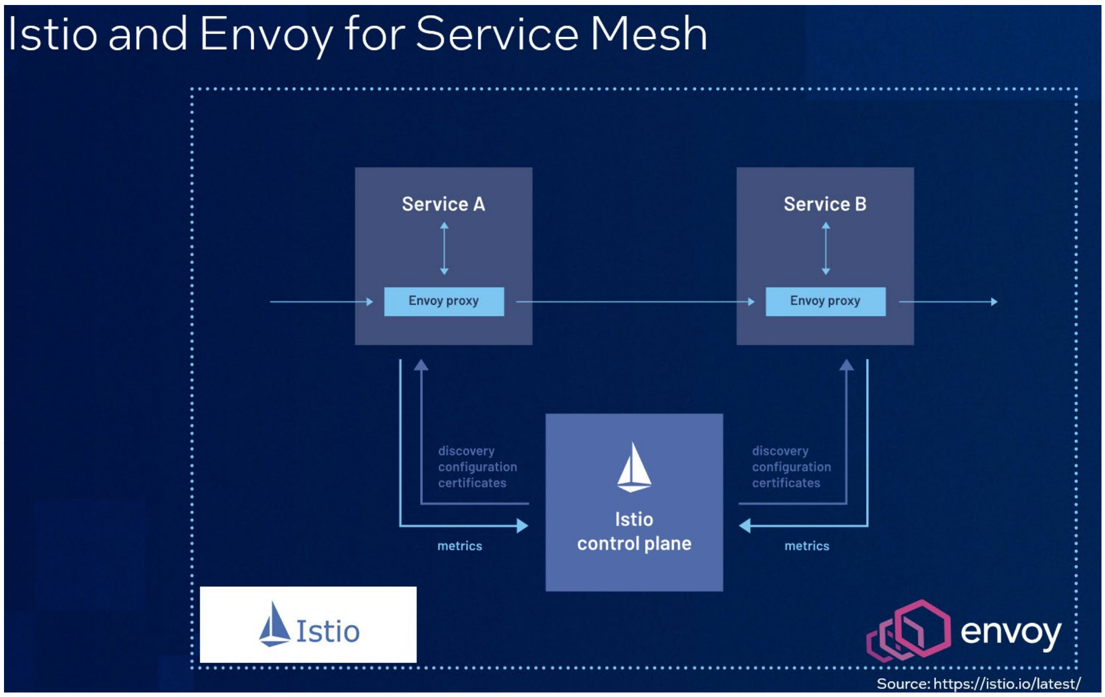


Figure 1. Istio and Envoy

Istio and Envoy Optimizations

Performance - TLS Accelerations using Intel® QAT and Intel® AVX-512

TLS Acceleration within Envoy can happen using Intel® QAT hardware acceleration or Intel® AVX-512 vectorized instruction set.

TLS Acceleration using Intel QAT

Crypto operations can be both symmetric and asymmetric in nature. Intel's optimizations implement the solution by using asynchronous TLS to take advantage of the hardware offload acceleration benefits, which also saves CPU cycles.

Intel AVX-512 utilizes Single Instruction Multiple Data (SIMD) vector instruction capabilities into the CPU. Recently, crypto instructions have been added to the vector instruction set Intel AVX-512. TLS handshake, when accelerated with Intel AVX-512, are executed in parallel and thus, improve performance.

Envoy uses BoringSSL as the default TLS library. BoringSSL supports setting private key methods for offloading asynchronous private key operations. Envoy implements a private key provider framework to allow creation of Envoy extensions, which handles TLS handshake private key operations (signing and decryption) using the BoringSSL hooks.

CryptoMB private key provider is an Envoy extension, which handles BoringSSL TLS RSA operations using Intel AVX-512 multi-buffer acceleration. When a new handshake occurs, BoringSSL invokes the private key provider to request the cryptographic operation, and then the control returns to Envoy. The RSA requests are gathered in a buffer. When the buffer is full or the timer expires, the private key provider invokes Intel AVX-512 processing of the buffer. When processing is done, Envoy is notified that the cryptographic operation is completed and that it may continue with the handshake.

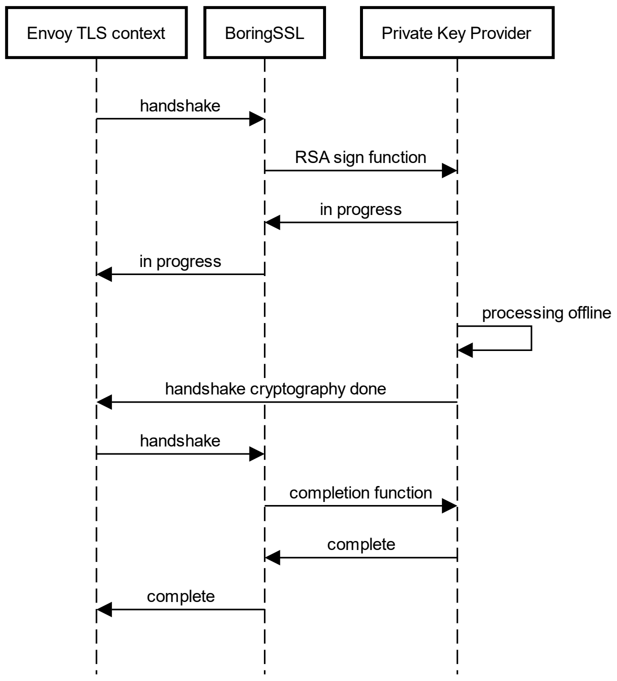


Figure 2. Envoy and Asynchronous Handshakes

The Envoy worker thread has a buffer size for eight RSA requests. When the first RSA request is stored in the buffer, a timer will be initiated (timer duration is set by the poll_delay field in the CryptoMB configuration). When the buffer is full or when the timer expires, perform the crypto operations for all RSA requests simultaneously. The SIMD processing gives the potential performance benefit compared to the non-accelerated case.

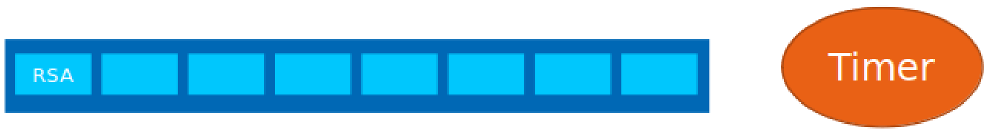


Figure 3: Crypto Multi-Buffer when Timer has started

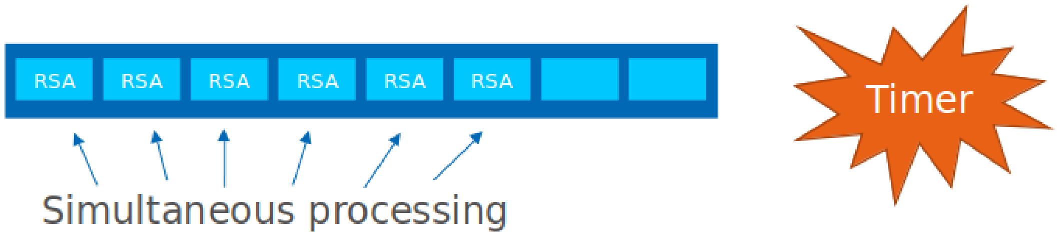


Figure 4. Crypto Multi-Buffer when Timer has expired

As illustrated in the figures above, when the CryptoMB timer expires there is a parallel execution of the crypto operations.

TLS Handshake Performance Improvement using Intel QAT

Intel QAT is a special hardware accelerator, which is visible to the operating system as a PCI device. The Envoy Intel QAT private key provider expects that the Intel QAT devices are available using the regular Linux kernel driver, present in Linux kernel from version 5.15 onward. The Intel QAT endpoint is exposed to Envoy via an SR-IOV VF device, which is the standard Intel QAT container deployment method, used, for example, in Kubernetes via Intel QAT device plugin.

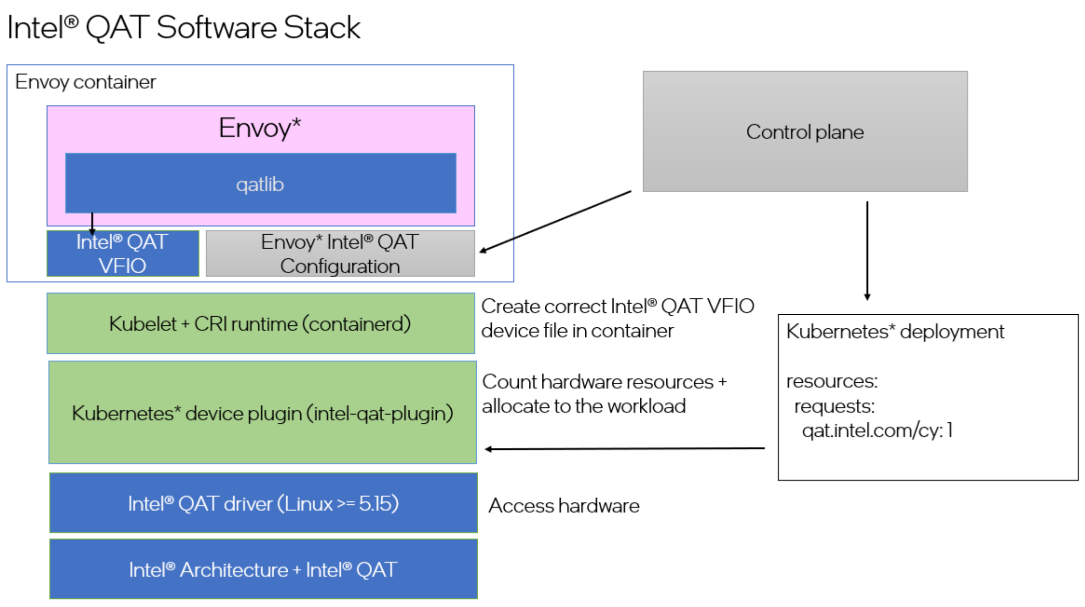

Intel® QAT Software Stack
Envoy container
Envoy*
qatlib
Intel® QAT VFIO
Envoy* Intel® QAT Configuration
Control plane
Kubelet + CRI runtime (containerd)
Create correct Intel® QAT VFIO device file in container
Kubernetes* deployment
resources:
requests:
qat.intel.com/cy: 1
Kubernetes* device plugin (intel-qat-plugin)
Count hardware resources + allocate to the workload
Intel® QAT driver (Linux >= 5.15)
Access hardware
Intel® Architecture + Intel® QAT

Figure 5. Envoy Intel QAT SW stack enablement for Kubernetes

The performance benefit from using Intel QAT for TLS handshakes depends on many factors. Most important is simply the amount of asymmetric cryptography that needs to be done because that makes the cryptography acceleration have more effect in the overall performance. For example, if there are only a few new TLS connections per second or if the selected RSA key size is small, the acceleration possibilities are smaller. Conversely, if the RSA key size is large and there are many incoming RSA connections, the possibility for performance increase is bigger. Another thing to consider is the number of CPU threads Envoy is running on. On smaller number of CPU cores the performance benefit is easier to see, since the acceleration leaves the CPU cores free to do other useful work needed for connection processing.

The performance impact has several components:

- The change in maximum throughput (requests / second)
- The change in latency (time required to complete a single operation)
- The change in CPU utilization
- The change in server power requirements

Performance – Intel® Dynamic Load Balancer (Intel® DLB) on 4th Gen Intel® Xeon® Scalable Processor

In Envoy, there are two main issues that Intel's DLB accelerator solves:

1. Distribution across cores is not even. Some cores are more occupied than the others.
2. Envoy has its exact balance SW load balancer, which balances connections, but CPU cycles utilized are uneven (20% in some, 80% in others)

Intel DLB offloads the distribution of requests across worker cores at the server.

The Intel DLB is a hardware managed system of queues and arbiters connecting producers and consumers. It is a PCI device in the CPU package. Intel DLB interacts with software running on cores and potentially other devices.

Intel DLB implements the load balancing features outlined earlier, including the following:

1. Lock-free multi-producer/multi-consumer operation
2. Multiple priorities for varying traffic types
3. Various distribution schemes

Data-plane software communicates with Intel DLB using standard (PCI) memory mapped interfaces in a simple, low cycle-cost way that is enabled with DPDK. Intel DLB supports virtualization using industry-standard techniques and is exposed as part of the Virtual Network Function Infrastructure on an Intel® architecture platform. Intel DLB further allows finer grained isolation between individual applications if necessary. Use Intel DLB to offload the distribution of requests across worker cores at the server.

More information on Intel DLB can be found at: https://builders.intel.com/docs/networkbuilders/SKU-343247-001US-queue-management-and-load-balancing-on-intel-architecture.pdf

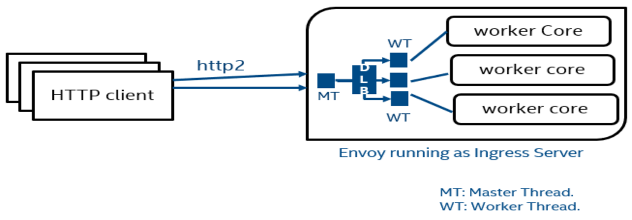


Figure 6: Solution based on Intel DLB

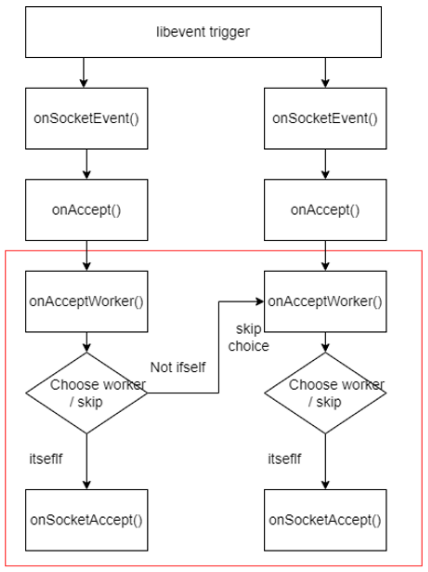


Figure 7. Traditional HTTP Traffic flow Handling in Envoy

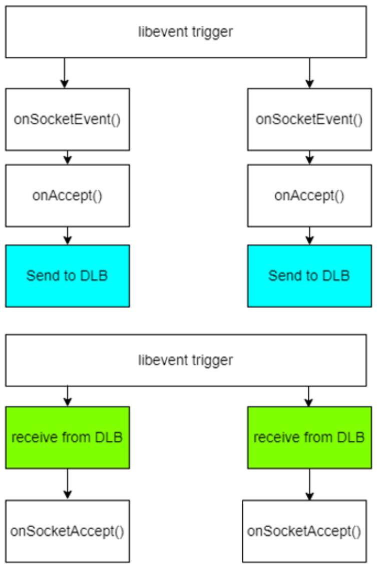


Figure 8. HTTP Traffic flow Handling with Intel DLB in Envoy

Performance – Envoy Routing Acceleration using Hyperscan

Envoy performs filtering, access control, and routing operations within a service mesh environment. During filtering operations, it selects different filters for different HTTP requests. For access control operations, Envoy utilizes access control to block suspicious requests by matching their characteristics with security policies. The routing actions involve parsing the URL paths and other components of HTTP requests that need to be routed upstream to different clusters or services.

During the above operations, regex matching is key factor that helps Envoy decide the routing of a request. These regex operations utilize expensive CPU cycles. Utilizing Hyperscan to optimize the regex matching operations improves the performance by saving CPU cycles and the latency of requests.

In operations mentioned above, matching is the basic but core module, which helps Envoy to decide where a request can be redirected, and the regex matching is one of the most expensive methods, which consumes much more CPU utilization compared to prefix and exact matching.

Performance - TCP/IP Bypass using eBPF

The current implementation of service mesh in Istio and Envoy involve an overhead of TCP/IP stack. Data plane implementation in Istio is through Envoy as a sidecar proxy. The data packets traverse the TCP/IP stack at least three times during the following situations:

- Inbound
- Outbound
- Envoy to Envoy within the same host

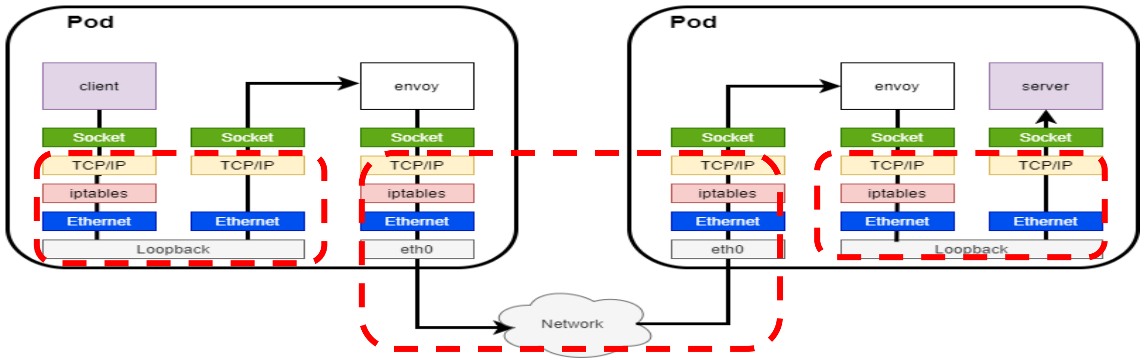


Figure 9. Typical Istio/Envoy Deployment within a K8's Cluster

The multiple TCP/IP stack traversal causes performance degradation in the data plane when envoy is deployed as a sidecar proxy. Intel's proposed solution is to bypass the TCP/IP networking stack in the Linux kernel by utilizing eBPF module. This solution has shown a reduction in latency and increase in the throughput because the data is being written directly to the socket in the user space.

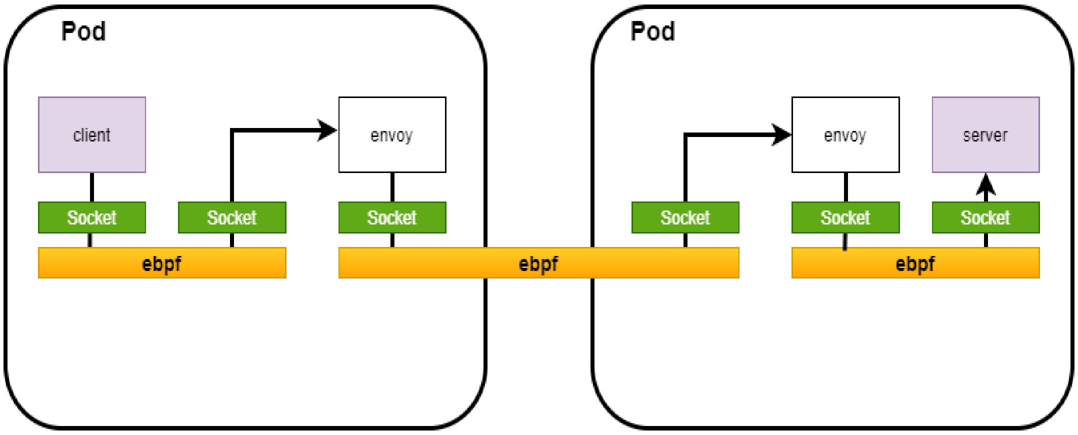


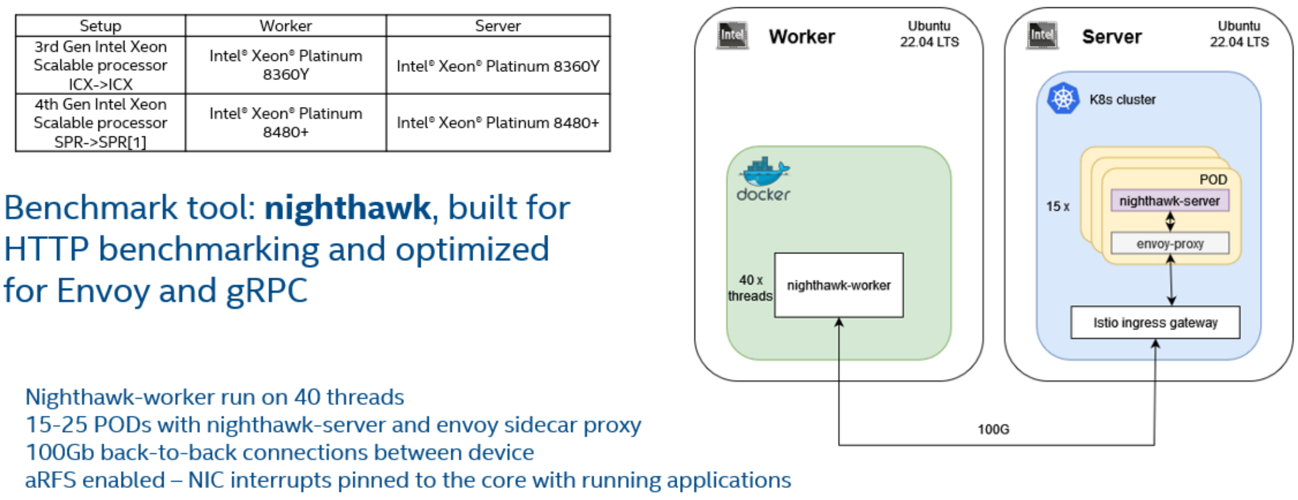
Figure 10. TCP/IP Bypass using eBPF

Benefits of Solution

Performance – TLS Accelerations on 4th Gen Intel Xeon Scalable Processor using Intel QAT and Intel AVX-512

TLS acceleration can be achieved on 4th Gen Intel Xeon Scalable processor using either Intel QAT accelerator or the Intel AVX-512 (Crypto MultiBuffer) vectorized acceleration. TLS acceleration using Intel QAT on 3rd Gen Intel® Xeon® Scalable processor is NOT supported.

The configuration is provided in Appendix 2.



Setup	Worker	Server
3rd Gen Intel Xeon Scalable processor ICX->ICX	Intel® Xeon® Platinum 8360Y	Intel® Xeon® Platinum 8360Y
4th Gen Intel Xeon Scalable processor SPR->SPR[1]	Intel® Xeon® Platinum 8480+	Intel® Xeon® Platinum 8480+

Benchmark tool: **nighthawk**, built for HTTP benchmarking and optimized for Envoy and gRPC

- Nighthawk-worker run on 40 threads
- 15-25 PODs with nighthawk-server and envoy sidecar proxy
- 100Gb back-to-back connections between device
- aRFS enabled – NIC interrupts pinned to the core with running applications

Figure 11. Benchmarking Setup

The performance and latency charts below showcase the TLS accelerations performance on 4th Gen Intel® Xeon® processor with AVX-512 (Crypto MultiBuffer), and different Intel QAT end points. The TLS accelerations can be achieved either with Intel QAT or Intel AVX-512.

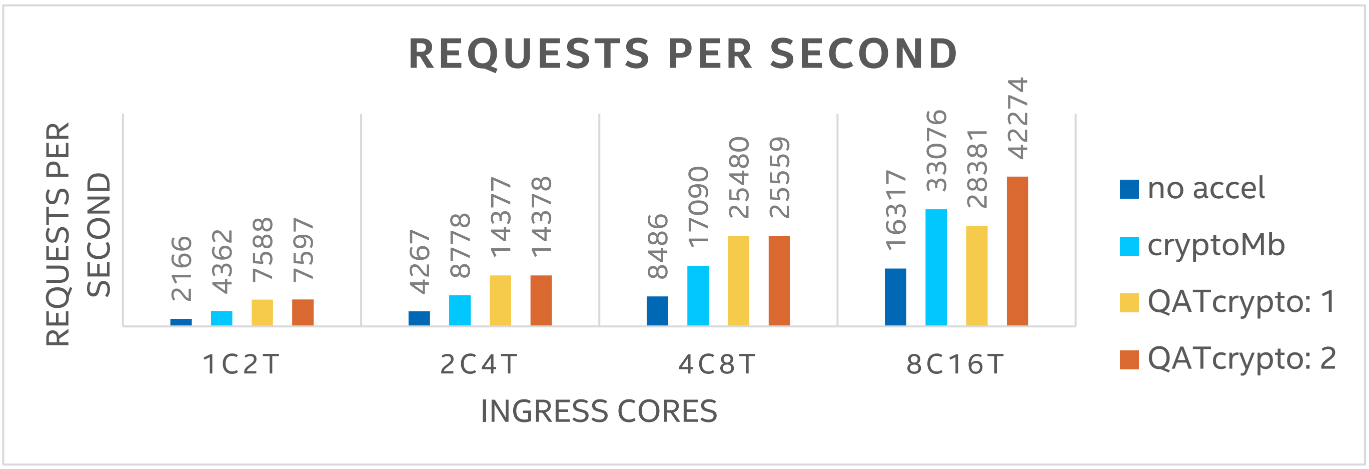
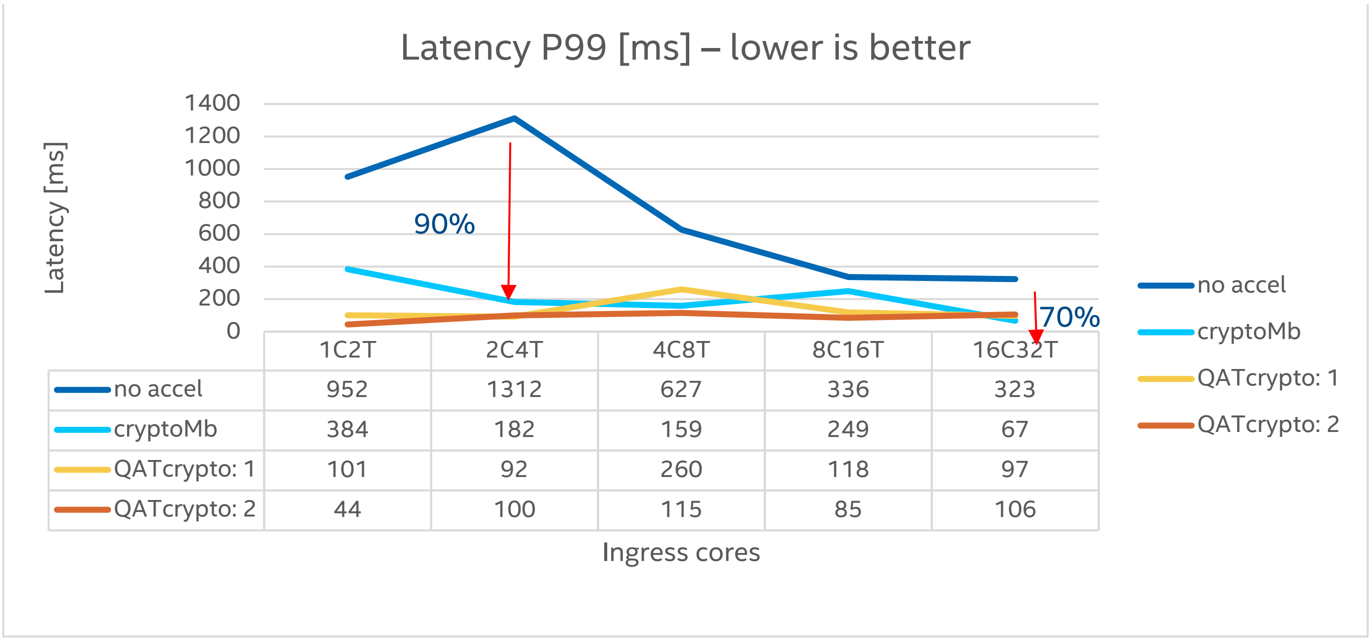


Figure 12. TLS Performance on 4th Gen Intel® Xeon® Scalable Processor



	1C2T	2C4T	4C8T	8C16T	16C32T
no accel	952	1312	627	336	323
cryptoMb	384	182	159	249	67
QATcrypto: 1	101	92	260	118	97
QATcrypto: 2	44	100	115	85	106

Figure 13. Latency on 4th Gen Intel Xeon Scalable Processor

Summary

Using 1x QAT, for 8 core and 16 cores can save 42% and 60% respectively on CPU cycles.

Using 2x QAT, for 8 core and 16 cores can save 18% and 49% for 2x QAT on CPU cycles. It is best to use 16 cores for 2x QAT.

Performance – Intel Dynamic Load Balancer (Intel DLB) on 4th Gen Intel Xeon Scalable Processor

The setup for the benchmarking is shown below:

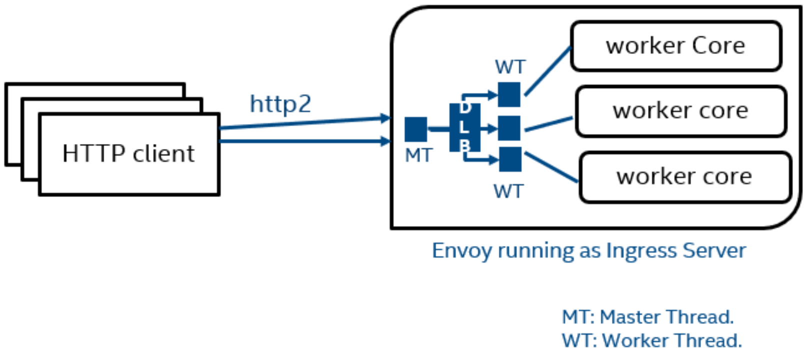


Figure 14: Intel DLB based solution in Envoy

Setup and configuration of the benchmark to showcase the Intel DLB benefits in Envoy using nighthawk is in Appendix 3.

Setup	Worker	Server
4th Gen Intel® Xeon® Scalable processor (SPR) ->SPR[1]	Intel® Xeon® Platinum 8480+	Intel® Xeon® Platinum 8480+

Nighthawk POD's with reponse size:
- 25 PODs with 1kB
- 25 PODs with 10kB
- 25 PODs with 1MB
- 100 PODS with mixed size
(25x1kB, 25x10kB, 25x100kB, 25x1MB)

- Istio ingress gateway working on to 6Cores/12Threads

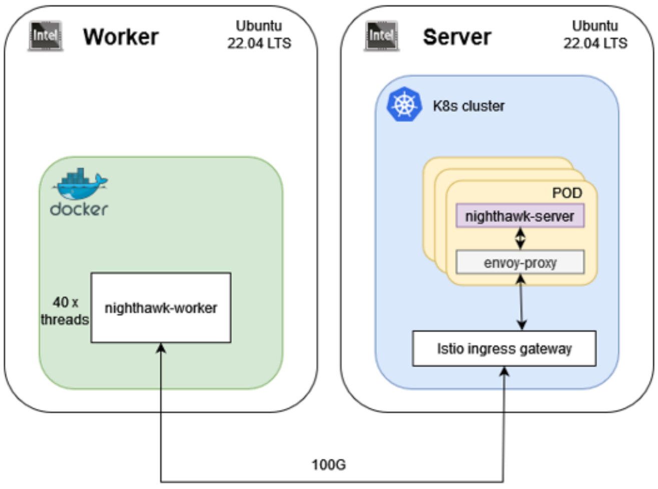


Figure 15: Benchmarking setup for Envoy optimization using Intel DLB

Performance

The latency of different message sizes with 4000 connections.

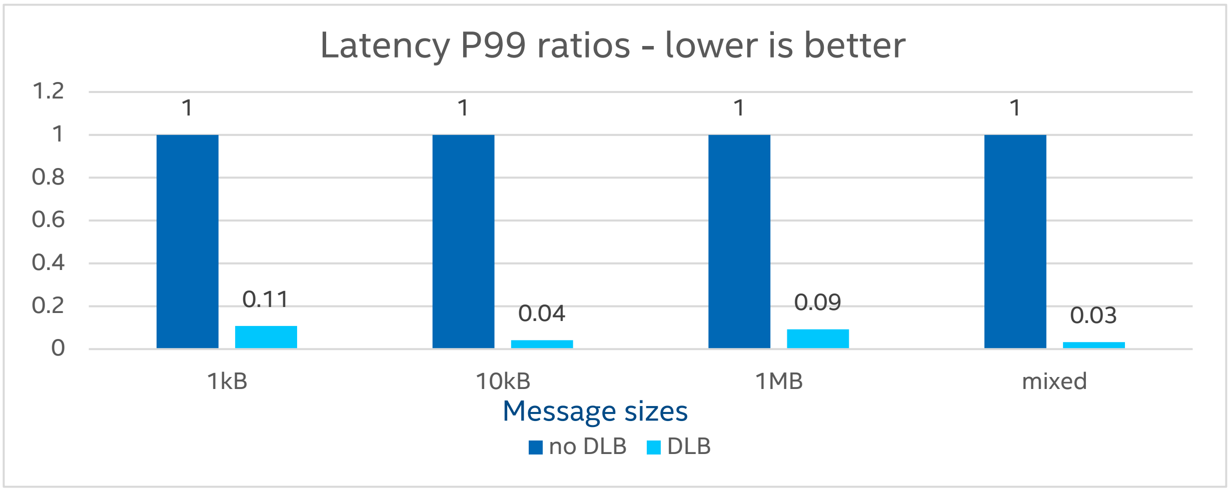


Figure 16: P99 Latency for different message sizes on 4 Gen Intel Xeon Scalable Processor

Summary

For the same RPS, latency improves upto 1.96x for mixed message sizes using Intel DLB in a 6C12Th setup.

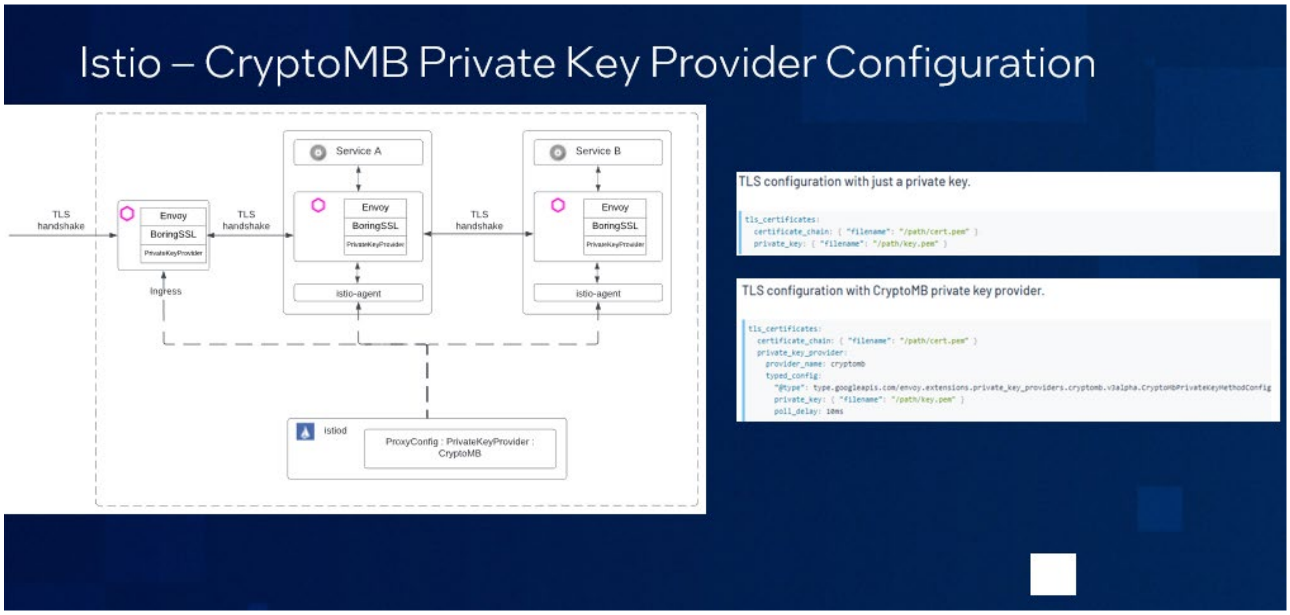
Use Case Examples

TLS Accelerations using Intel AVX-512 and Intel QAT

Intel® AVX-512

An example deployment below shows a sample configuration of Istio with Envoy as a sidecar proxy using Intel AVX-512 vectorized instruction set. Envoy is deployed as an Ingress Gateway within the cluster whereas there are two microservices running Envoy as a sidecar proxy within a Kubernetes cluster. A user can apply the private key provider configuration to:

1. Ingress Gateway only
2. Application Specific Pods by configuring them using pod annotations



TLS configuration with just a private key.

```
tls_certificates:
  certificate_chain: { "filename": "/path/cert.pem" }
  private_key: { "filename": "/path/key.pem" }
```

TLS configuration with CryptoMB private key provider.

```
tls_certificates:
  certificate_chain: { "filename": "/path/cert.pem" }
  private_key_provider:
    provider_name: cryptomb
    typed_config:
      "@type": type.googleapis.com/envoy.extensions.private_key_providers.cryptomb.v3alpha.CryptoMbPrivateKeyMethodConfig
      private_key: { "filename": "/path/key.pem" }
      poll_delay: 10ms
```

Figure 17. Istio Deployment utilizing Intel AVX-512

TLS Configuration with only a private key

```
tls_certificates:
  certificate_chain: { "filename": "/path/cert.pem" }
  private_key: { "filename": "/path/key.pem" }
```

TLS Configuration with CryptoMB private key provider

```
tls_certificates:
  certificate_chain: { "filename": "/path/cert.pem" }
  private_key_provider:
    provider_name: cryptomb
    typed_config:
      "@type":
type.googleapis.com/envoy.extensions.private_key_providers.cryptomb.v3alpha.CryptoMbPrivate
KeyMethodConfig
      private_key: { "filename": "/path/key.pem" }
      poll_delay: 10ms
```

Intel® QAT

Envoy supports Intel QAT for accelerating TLS handshakes. The performance benefit varies depending on the use case, but Intel QAT can help in reducing CPU usage, reducing individual request latency, and increasing throughput. The Intel QAT to Envoy support needs to be enabled by a configuration file change or by a dynamic Envoy listener configuration over the xDS protocol. In addition, the Envoy container must have Intel QAT resources added to it by configuring Kubernetes cluster accordingly.

Envoy TLS configuration can be done by two methods: either using direct configuration from a configuration file or using SDS (Secret Discovery Service) protocol for remotely configuring Envoy from an external control plane. Intel QAT TLS acceleration can be enabled in both ways.

When using direct configuration file configuration, the regular way for setting the private key is by adding it to as private_key field to Envoy's common_tls_context [1]:

```
common_tls_context:
  tls_certificates:
  - certificate_chain:
      filename: "/tmp/rsa-cert.pem"
    private_key:
      filename: "/tmp/rsa-key.pem"
```

However, when Intel QAT acceleration is required, private_key field should be replaced with suitably configured private_key_provider field:

```
common_tls_context:
  tls_certificates:
  - certificate_chain:
      filename: "/tmp/rsa-cert.pem"
    private_key_provider:
      provider_name: qat
      typed_config:
        "@type":
"type.googleapis.com/envoy.extensions.private_key_providers.qat.v3alpha.QatPrivateKeyMethod
Config"
         poll_delay: 0.002s
         private_key:
           filename: "/tmp/rsa-key.pem"
```

The Intel QAT private key provider configuration has two fields: poll_delay and private_key. The private_key field works as a regular Envoy DataSource type. The poll_delay field is a Duration type and specifies how often the Intel QAT instance should be polled when waiting for an answer to Intel QAT request. The right value depends on the tradeoff between CPU consumption and latency requirements and might require experimentation depending on the workload setup. A value of 0.002s (2 milliseconds) is a good starting point.

Intel Dynamic Load Balancer (Intel DLB) on 4th Gen Intel Xeon Scalable Processor

The most prominent use case is when Envoy is used as an Ingress Proxy Server. Deploying Intel DLB solution show cases the benefits.

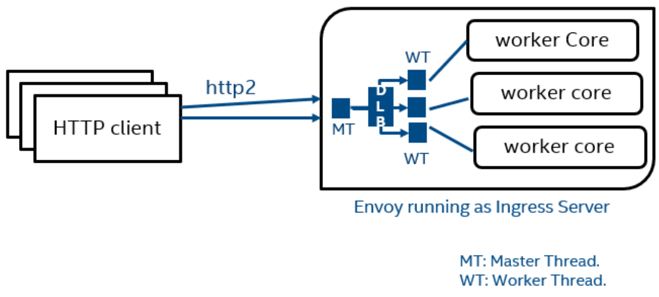


Figure 18. Envoy as an Ingress Proxy Server with Intel DLB optimizations

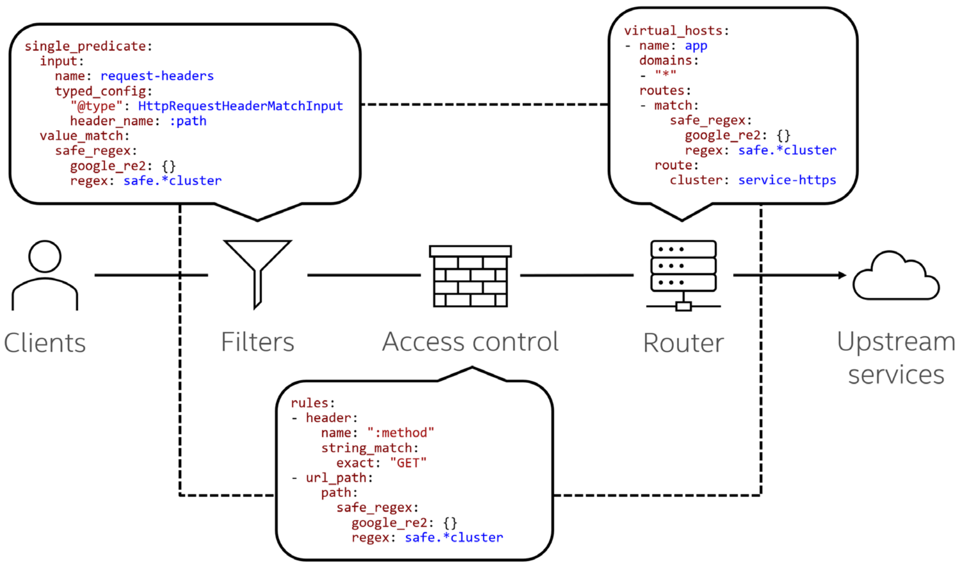
An example configuration:

```
static_resources:
  listeners:
  - connection_balance_config:
      extend_balance:
        name: envoy.network.connection_balance.dlb
        typed_config:
          "@type": type.googleapis.com/envoy.extensions.network.connection_balance.dlb.v3alpha.Dlb
```

Installation and Configuration details of the driver are listed here.

Envoy Routing Acceleration using Hyperscan

This use case describes when an Envoy is used as an Ingress Gateway. The following diagram illustrates the different filters and steps that are triggered when Envoy must make routing decisions.



```
single_predicate:
  input:
    name: request-headers
    typed_config:
      "@type": HttpRequestHeaderMatchInput
      header_name: :path
  value_match:
    safe_regex:
      google_re2: {}
      regex: safe.*cluster
```

```
virtual_hosts:
- name: app
  domains:
  - "*"
  routes:
  - match:
      safe_regex:
        google_re2: {}
        regex: safe.*cluster
    route:
      cluster: service-https
```

```
rules:
- header:
    name: ":method"
    string_match:
      exact: "GET"
- url_path:
    path:
      safe_regex:
        google_re2: {}
        regex: safe.*cluster
```

Figure 19. Envoy routing flow for incoming requests

TCP/IP Bypass using eBPF

The below use case is an example of deploying eBPF within service mesh.

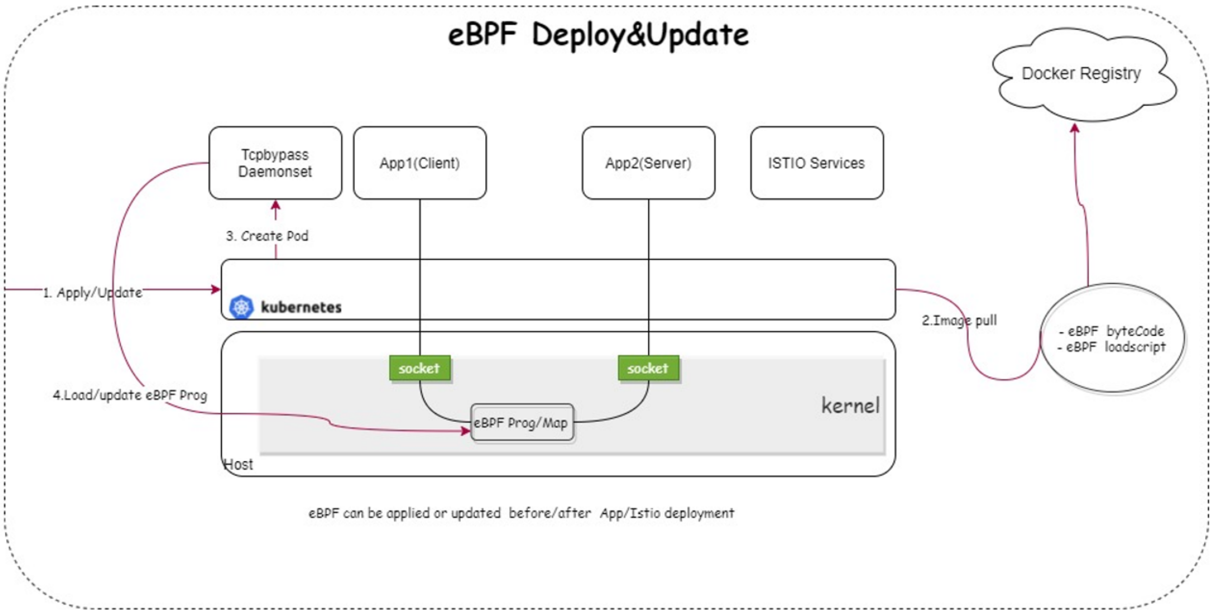
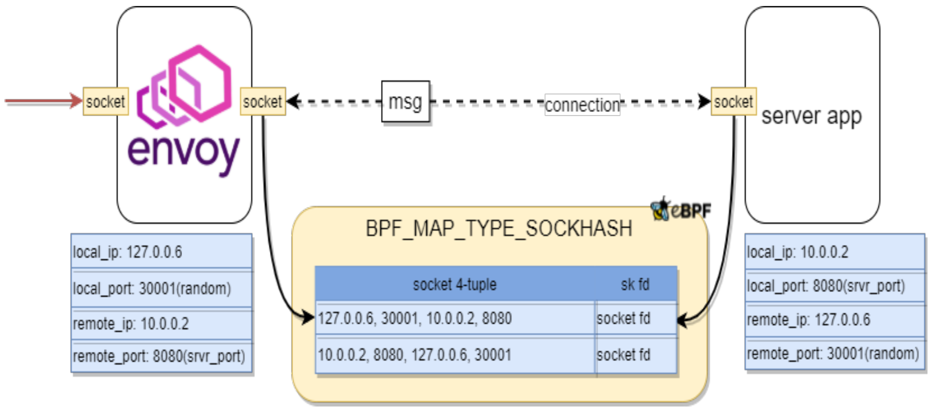


Figure 20. Using eBPF Solution within Service Mesh Envoy Deployment

Inbound Acceleration

For inbound, server-side envoy connects server application actively. Envoy will hold an active socket, and server app holds a passive socket. SOCK_OPS is used to define and insert the callback function to kernel; therefore, corresponding callback will be invoked when TCP state changes. When active socket hits the active established state and passive socket hits passive established state respectively, socket 4-tuple address and socket FD of these two sockets are recorded to the SOCKHASH map. After TCP handshake is done and the socket tries to send message, its peer socket can be looked up from the map based on peer socket 4-tuple address by reversing the local and remote address. This design requires Istio version greater than v1.10, which has the flag INBOUND_PASSTHROUGH set by default. This flag makes Envoy to use pod IP as the destination IP to communicate with server app.. If this flag is not set, Envoy will use localhost IP that will cause conflict when populating the map since our SOCK_OPS program is attached to unified cgroup.



local_ip: 127.0.0.6
local_port: 30001(random)
remote_ip: 10.0.0.2
remote_port: 8080(srvr_port)

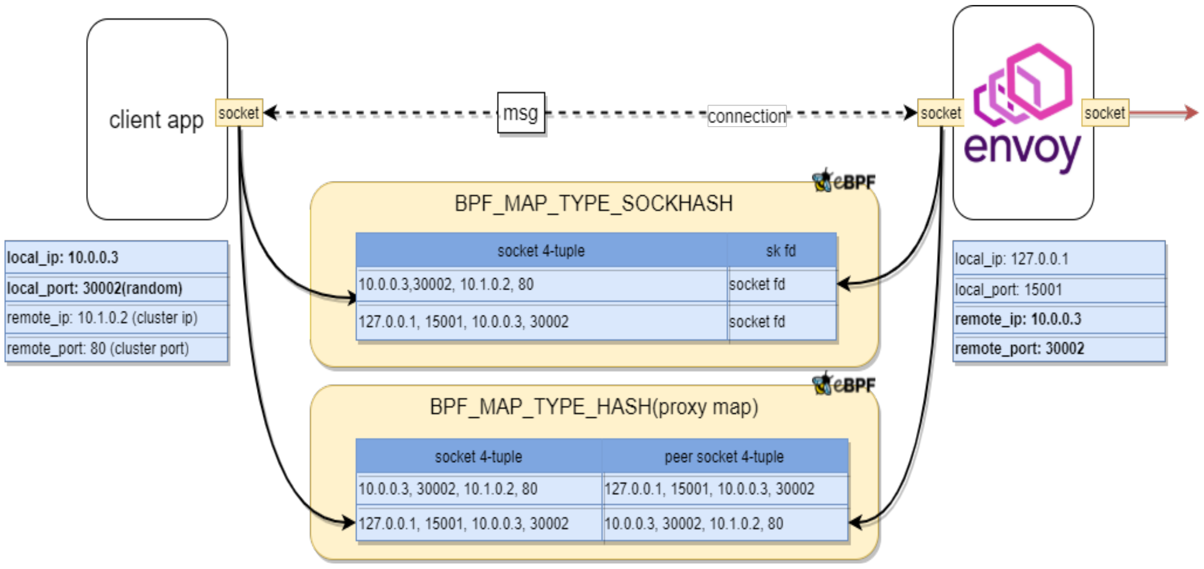
socket 4-tuple	sk fd
127.0.0.6, 30001, 10.0.0.2, 8080	socket fd
10.0.0.2, 8080, 127.0.0.6, 30001	socket fd

local_ip: 10.0.0.2
local_port: 8080(srvr_port)
remote_ip: 127.0.0.6
remote_port: 30001(random)

Figure 21. TCP/IP eBPF Bypass Acceleration for Inbound Traffic

Outbound Acceleration

For outbound, client application tries connecting to another service. After iptables rule is applied, the traffic is redirected to Envoy and client application sets up a TCP connection with the sidecar. So, client app holds the active socket, which has cluster IP and port as its remote address, and Envoy holds the passive socket, which has localhost IP and 15001 as local address. Recording the socket 4-tuple address and socket FD (like in the case of inbound acceleration) is not enough here to find out peer socket. To resolve this problem, a proxy map is introduced to record the socket pair addresses during TCP handshake. Source address never changes during communication, it helps to identify which two sockets belong to the same connection.



local_ip: 10.0.0.3
local_port: 30002(random)
remote_ip: 10.1.0.2 (cluster ip)
remote_port: 80 (cluster port)

socket 4-tuple	sk fd
10.0.0.3,30002, 10.1.0.2, 80	socket fd
127.0.0.1, 15001, 10.0.0.3, 30002	socket fd

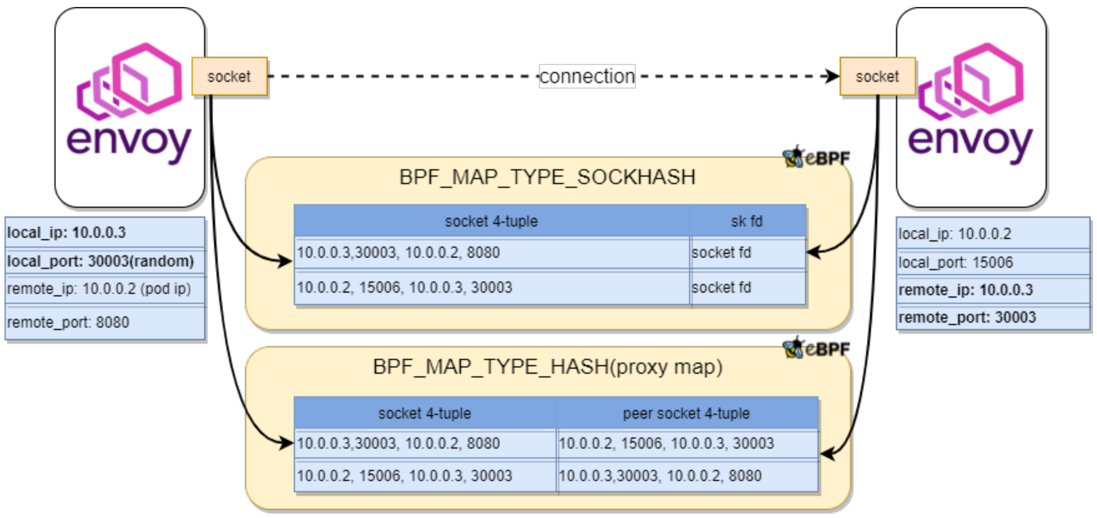
local_ip: 127.0.0.1
local_port: 15001
remote_ip: 10.0.0.3
remote_port: 30002

socket 4-tuple	peer socket 4-tuple
10.0.0.3, 30002, 10.1.0.2, 80	127.0.0.1, 15001, 10.0.0.3, 30002
127.0.0.1, 15001, 10.0.0.3, 30002	10.0.0.3, 30002, 10.1.0.2, 80

Figure 22. TCP/IP eBPF Bypass Acceleration for Outbound Traffic

Envoy to Envoy Acceleration on the Same Host

Envoy-to-Envoy acceleration is almost the same as outbound. This is because, from the perspective of kernel, they are all redirected by iptables, and destination address is modified. If the two envoys are on different hosts then the proxy map will not be populated because two hosts cannot share the map content. The mapping between original destination and new address cannot be established on a single host. Therefore, SK_MSG cannot match any entry to get peer socket address in the proxy map when sending message. In this cross-node case, package is sent out through TCP/IP stack.



local_ip: 10.0.0.3
local_port: 30003(random)
remote_ip: 10.0.0.2 (pod ip)
remote_port: 8080

socket 4-tuple	sk fd
10.0.0.3,30003, 10.0.0.2, 8080	socket fd
10.0.0.2, 15006, 10.0.0.3, 30003	socket fd

local_ip: 10.0.0.2
local_port: 15006
remote_ip: 10.0.0.3
remote_port: 30003

socket 4-tuple	peer socket 4-tuple
10.0.0.3,30003, 10.0.0.2, 8080	10.0.0.2, 15006, 10.0.0.3, 30003
10.0.0.2, 15006, 10.0.0.3, 30003	10.0.0.3,30003, 10.0.0.2, 8080

Figure 23. TCP/IP eBPF Bypass Acceleration between Envoy side-cars on same host

Summary

Service Mesh has introduced latency and performance challenges due to the nature of sidecar implementation in Envoy. Intel has addressed the performance and latency challenges by utilizing Intel Xeon features such as Intel QAT, Intel DLB. The results of performance and latency improvements can be summarized as below.

- Up to 1.6x CPU cycles saved, up to 2.37x throughput/RPS improvement and up to 1.95x latency reduction using Intel QAT in a 1C-16C scaling experiment on 4th Gen Intel Xeon Scalable processor
- Using 1x Intel QAT, for 8 core and 16 cores, can save 42% and 60% respectively on 4th Gen Intel Xeon Scalable processor
- Using 2x Intel QAT, for 8 core and 16 cores, can save 18% and 49% for 2x Intel QAT on 4th Gen Intel Xeon Scalable processor
- For the same RPS, latency improves upto 1.96x for mixed message sizes using Intel DLB on 4th Gen Intel Xeon Scalable processor

Availability of Performance Solutions

Table 1: Availability of Performance Solutions

Performance Optimization	Istio Availability	Envoy Version	Intel GitHub	Dependency	Intel Platforms
Intel® QAT TLS Performance	1.17 – Available in 2023.	1.24	Intel GitHub (Available)	QATlib	4th Gen Intel Xeon Scalable processor
Intel® Crypto MultiBuffer Perf	1.14	1.20	Intel GitHub (Available)	CryptoMB Library	4th Gen Intel Xeon Scalable processor, 3rd Gen Intel Xeon Scalable processor
Intel® DLB Performance	1.18 – Available in 2023.	1.23	Intel GitHub (Available)	libDLB, Drivers.	4th Gen Intel Xeon Scalable processor
Intel® Hyperscan	1.14	1.22	Intel GitHub (Available)	Hyperscan	4th Gen Intel Xeon Scalable processor, 3rd Gen Intel Xeon Scalable processor
Intel® TCP/IP eBPF Bypass	N/A	N/A	Intel GitHub (Available)	None	4th Gen Intel Xeon Scalable processor, 3rd Gen Intel Xeon Scalable processor

Terminology

Table 2. Terminology

Abbreviation	Description
CPU	Central processing unit
eBPF	Extended Berkley Packet Filter
Hyperscan	Hyperscan is a high-performance multiple regex matching library
Intel® AVX-512	Intel® Advanced Vector Extensions 512
Intel® DLB	Intel® Dynamic Load Balancer
Intel® QAT	Intel® QuickAssist Technology
RSA	Rivest–Shamir–Adleman – A public-key cryptosystem.
SIMD	Single Instruction Multiple Data
TCP/IP	Transmission Control Protocol/Internet Protocol
TLS	Transport Layer Security

References

Table 3. References

Reference	Source
CryptoMB - TLS handshake acceleration for Istio	https://istio.io/latest/blog/2022/cryptomb-privatekeyprovider/
DLB Envoy	https://www.envoyproxy.io/docs/envoy/latest/configuration/other_features/dlb
Service Mesh - Crypto Accelerations in Istio and Envoy with Intel Xeon Scalable Processors User Guide	https://networkbuilders.intel.com/solutionslibrary/service-mesh-crypto-accelerations-istio-envoy-intel-xeon-sp-user-guide
Service Mesh - Envoy Regular Expression Matching Acceleration with Hyperscan User Guide	https://networkbuilders.intel.com/solutionslibrary/service-envoy-regular-expression-matching-acceleration-hyperscan-user-guide
Service Mesh – mTLS Key Management in Istio and Envoy for Intel® Xeon® Scalable Processors User Guide	https://networkbuilders.intel.com/solutionslibrary/service-mesh-mtls-key-mgmt-istio-envoy-intel-xeon-sp-user-guide
Service Mesh - TCP/IP eBPF Bypass in Istio and Envoy with Intel Xeon Scalable Processors User Guide	https://networkbuilders.intel.com/solutionslibrary/service-mesh-tcp-ip-bypass-istio-envoy-intel-xeon-sp-user-guide

Document Revision History

Revision	Date	Description
001	January 2023	Initial release.

Appendix 1: Performance Improvements Benchmarking on Intel® QAT, Intel® AVX-512, and Intel® DLB

Software Configuration	3rd Gen Intel Xeon Scalable processor Ice lake[1]	3rd Gen Intel Xeon Scalable processor Ice Lake[2]	4th Gen Intel Xeon Scalable processor Sapphire Rapids – SPR[1] E3	4th Gen Intel Xeon Scalable processor Sapphire Rapids – SPR[2] (with QAT)
Workload & version	Nighthawk	Nighthawk	Nighthawk	Nighthawk
Compiler	gcc version 11.2.0 (Ubuntu 11.2.0-19ubuntu1)	gcc version 11.2.0 (Ubuntu 11.2.0-19ubuntu1)	gcc version 11.2.0 (Ubuntu 11.2.0-19ubuntu1)	gcc version 11.2.0 (Ubuntu 11.2.0-19ubuntu1)
Libraries				
OS	Ubuntu 22.04 LTS	Ubuntu 22.04 LTS	Ubuntu 22.04 LTS	Ubuntu 22.04 LTS
Kernel	5.15.0-39-generic	5.15.0-50-generic	5.15.0-40-generic	5.17.0-051700-generic
Docker	20.10.17	20.10.17	20.10.17	20.10.17
Kubernetes	v1.22.3	v1.22.3	v1.22.3	v1.22.3
Istio	1.13.4	1.13.4	1.13.4	Intel 22.06(1.14)
Calico	3.21.4	3.21.4	3.21.4	3.21.4
Run Method:	Warm	Warm	Warm	Warm
Iterations and result choice (median, average, min, max)	3 iterations, max	3 iterations, max	3 iterations, max	3 iterations, max
Protocol	HTTP/1.1 and HTTP/2	HTTP/1.1 and HTTP/2	HTTP/1.1 and HTTP/2	HTTPS
Payload size	400B	400B	400B	400B
Client threads	40	40	40	40
Operating Frequency	2.0GHz	2.4GHz	2.0GHz	2.0GHz
MTU	1500	1500	1500	1500
aRFS	enabled	enabled	enabled	enabled

Idle-poll	disabled	disabled	disabled	disabled

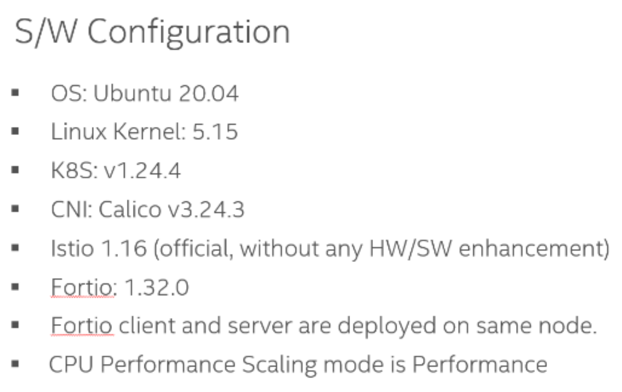
Appendix 2: TLS Accelerations using Intel® QAT and Intel® AVX-512 Configuration

Configuration	4th Gen Intel Xeon Scalable processor
Sockets	2
CPU Frequency	2.0GHz
Uncore Frequency	2.0GHz
RAM	1024GB [4400 MT/s]
Message size	400B

Appendix 3: Intel® DLB Configuration

	4th Gen Intel Xeon Scalable processor
Sockets	2
CPU Frequency	2.0GHz
Uncore Frequency	2.0GHz
RAM	1024GB [4400 MT/s]
4	1kB, 10kB, 1MB

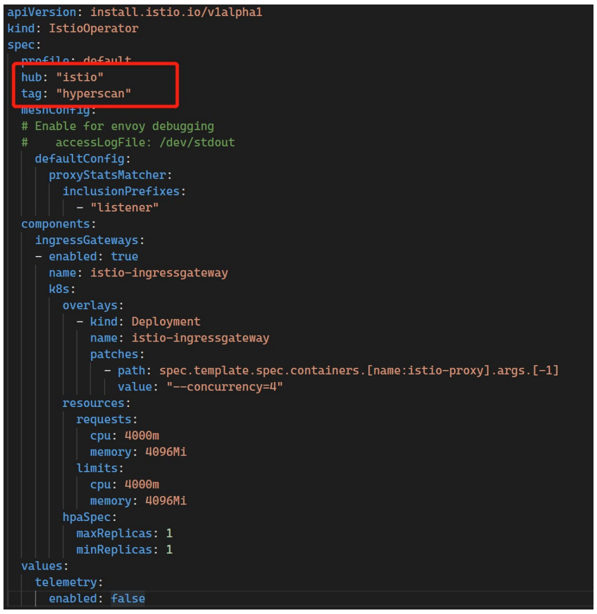
Appendix 4: Hyperscan Benchmarking Configuration



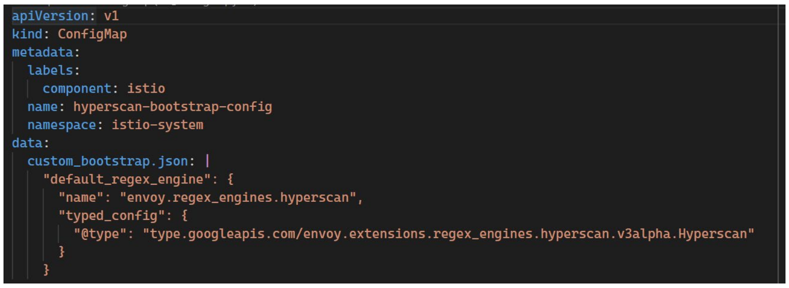
S/W Configuration

- OS: Ubuntu 20.04
- Linux Kernel: 5.15
- K8S: v1.24.4
- CNI: Calico v3.24.3
- Istio 1.16 (official, without any HW/SW enhancement)
- Fortio: 1.32.0
- Fortio client and server are deployed on same node.
- CPU Performance Scaling mode is Performance

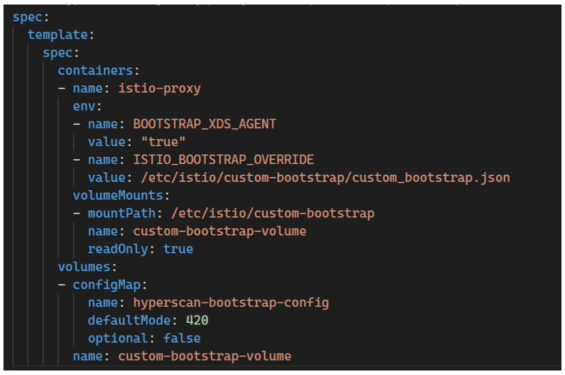
- Version 1.16
- One Istio ingress gateway with 4 CPU cores allocated
- Need to build Istio proxyv2 image manually, because Hyperscan is disabled by default in Istio Proxy
- Use a Kubernetes ConifgMap to pass the Envoy Regex Engine bootstrap configuration to enable Hyperscan Regex Engine in Istio Ingressgateway, default Regex Engine is Google RE2
- Use 100 RBAC Rules to match request's headers via EnvoyFilters



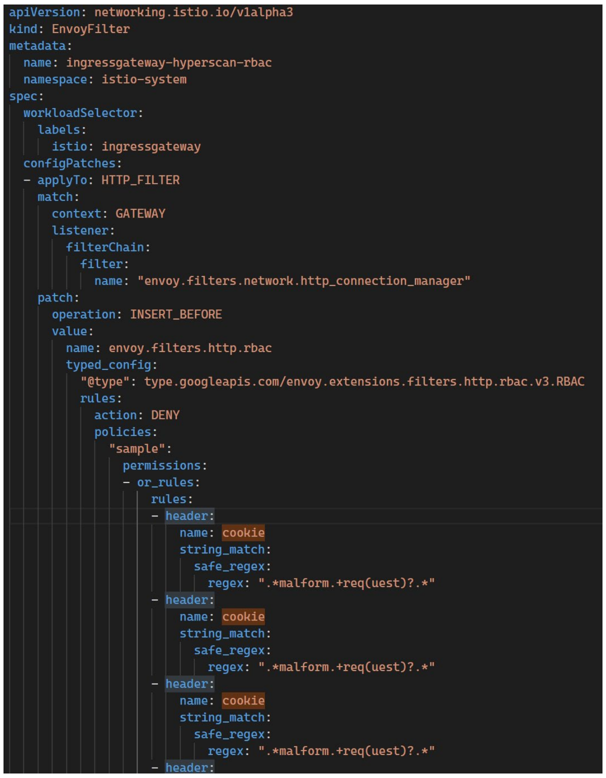
```yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  profile: default
  hub: "istio"
  tag: "hyperscan"
  meshConfig:
  # Enable for envoy debugging
  #    accessLogFile: /dev/stdout
    defaultConfig:
      proxyStatsMatcher:
        inclusionPrefixes:
          - "listener"
  components:
    ingressGateways:
    - enabled: true
      name: istio-ingressgateway
      k8s:
        overlays:
          - kind: Deployment
            name: istio-ingressgateway
            patches:
              - path: spec.template.spec.containers.[name:istio-proxy].args.[-1]
                value: "--concurrency=4"
        resources:
          requests:
            cpu: 4000m
            memory: 4096Mi
          limits:
            cpu: 4000m
            memory: 4096Mi
        hpaSpec:
          maxReplicas: 1
          minReplicas: 1
  values:
    telemetry:
      enabled: false
```



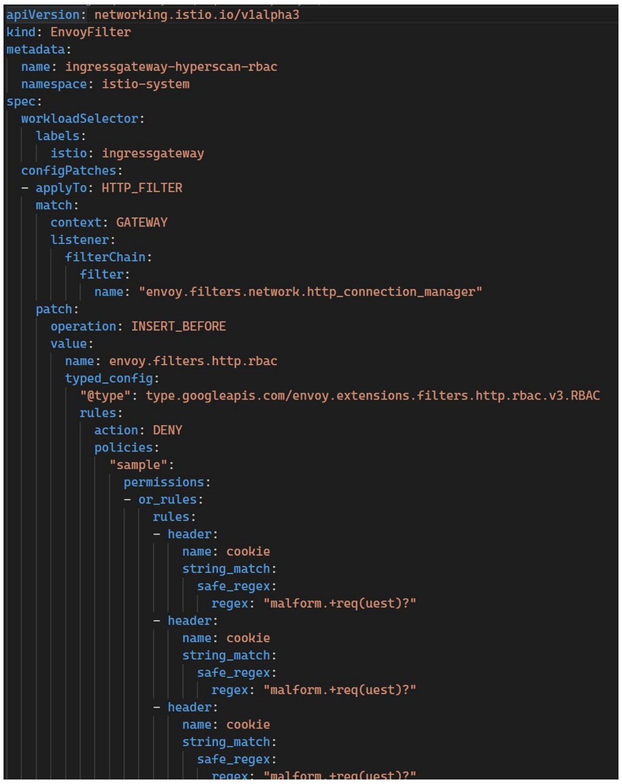
```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  labels:
    component: istio
  name: hyperscan-bootstrap-config
  namespace: istio-system
data:
  custom_bootstrap.json: |
    "default_regex_engine": {
      "name": "envoy.regex_engines.hyperscan",
      "typed_config": {
        "@type": "type.googleapis.com/envoy.extensions.regex_engines.hyperscan.v3alpha.Hyperscan"
      }
    }
```



```yaml
spec:
  template:
    spec:
      containers:
      - name: istio-proxy
        env:
        - name: BOOTSTRAP_XDS_AGENT
          value: "true"
        - name: ISTIO_BOOTSTRAP_OVERRIDE
          value: /etc/istio/custom-bootstrap/custom_bootstrap.json
        volumeMounts:
        - mountPath: /etc/istio/custom-bootstrap
          name: custom-bootstrap-volume
          readOnly: true
      volumes:
      - configMap:
          name: hyperscan-bootstrap-config
          defaultMode: 420
          optional: false
        name: custom-bootstrap-volume
```



```yaml
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: ingressgateway-hyperscan-rbac
  namespace: istio-system
spec:
  workloadSelector:
    labels:
      istio: ingressgateway
  configPatches:
  - applyTo: HTTP_FILTER
    match:
      context: GATEWAY
      listener:
        filterChain:
          filter:
            name: "envoy.filters.network.http_connection_manager"
    patch:
      operation: INSERT_BEFORE
      value:
        name: envoy.filters.http.rbac
        typed_config:
          "@type": type.googleapis.com/envoy.extensions.filters.http.rbac.v3.RBAC
          rules:
            action: DENY
            policies:
              "sample":
                permissions:
                - or_rules:
                    rules:
                    - header:
                        name: cookie
                        string_match:
                          safe_regex:
                            regex: ".*malform.+req(uest)?.*"
                    - header:
                        name: cookie
                        string_match:
                          safe_regex:
                            regex: ".*malform.+req(uest)?.*"
                    - header:
                        name: cookie
                        string_match:
                          safe_regex:
                            regex: ".*malform.+req(uest)?.*"
                    - header:
```



```yaml
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: ingressgateway-hyperscan-rbac
  namespace: istio-system
spec:
  workloadSelector:
    labels:
      istio: ingressgateway
  configPatches:
  - applyTo: HTTP_FILTER
    match:
      context: GATEWAY
      listener:
        filterChain:
          filter:
            name: "envoy.filters.network.http_connection_manager"
    patch:
      operation: INSERT_BEFORE
      value:
        name: envoy.filters.http.rbac
        typed_config:
          "@type": type.googleapis.com/envoy.extensions.filters.http.rbac.v3.RBAC
          rules:
            action: DENY
            policies:
              "sample":
                permissions:
                - or_rules:
                    rules:
                    - header:
                        name: cookie
                        string_match:
                          safe_regex:
                            regex: "malform.+req(uest)?"
                    - header:
                        name: cookie
                        string_match:
                          safe_regex:
                            regex: "malform.+req(uest)?"
                    - header:
                        name: cookie
                        string_match:
                          safe_regex:
                            regex: "malform.+req(uest)?"
```

0123/DN/WIT/PDF 763381-001US