User Guide

# Service Mesh - Crypto Accelerations in Istio and Envoy with Intel® Xeon® Scalable Processors


## Authors

Ismo Puustinen

Mikko Ylinen

Ramesh Masavarapu


## 1 Introduction

Transport Layer Security (TLS) uses both asymmetric and symmetric cryptography. Handshakes are done using asymmetric public key algorithms (RSA and ECDSA). During the handshake, both ends agree on a key. The key is then used to encrypt/decrypt the communication using symmetric cryptography (AES).

Cryptographic operations can be slow. RSA is known to be computationally expensive. This is a known issue in edge ingress gateways, which handles thousands of new connections per second.

Cryptographic operations, both symmetric (AES) and asymmetric (RSA), can be accelerated using Intel® QuickAssist Technology (Intel® QAT) or Intel® Advanced Vector Extensions 512 (Intel® AVX-512) instructions. Intel® QAT is a special hardware accelerator, which is visible to the operating system as a PCI device. When using Intel® AVX-512 for acceleration, the acceleration happens by using SIMD (Single Instruction, Multiple Data) instructions. This is the reason that the acceleration is often referred to as “Multi Buffer”. In Envoy-OpenSSL, both acceleration modes are available, but in Envoy-BoringSSL (which, is the default in Istio) only asymmetric encryption acceleration can be used. This document only describes Envoy with BoringSSL. Optimizations on Envoy with OpenSSL is currently not supported and is expected to be supported in 2023.

The Intel AVX-512 instructions are available in the recent 3rd Gen Intel® Xeon® Scalable processors and newer. The Intel QuickAssist Technology (Intel QAT) are only available on 4th Gen Intel® Xeon® Scalable processors.

This document is part of the Network Transformation Experience Kits.

# Table of Contents



# Figures



# Tables



# Document Revision History

| Revision | Date | Description |
|---|---|---|
| 001 | January 2023 | Initial release. |

## 1.1 Terminology

Table 1. Terminology

| Abbreviation | Description |
|---|---|
| CPU | Central processing unit |
| ECDSA | Elliptic Curve Digital Signature Algorithm |
| Intel® AVX-512 | Intel® Advanced Vector Extensions 512 |
| Intel® QAT | Intel® QuickAssist Technology |
| PCI | Peripheral Component Interconnect |
| RSA | Rivest–Shamir–Adleman – A public-key cryptosystem |
| SIMD | Single Instruction Multiple Data |
| TCP/IP | Transmission Control Protocol/Internet Protocol |
| TLS | Transport Layer Security |

## 1.2 Reference Documentation

Table 2. Reference Documents

| Reference | Source |
|---|---|
| CryptoMB - TLS handshake acceleration for Istio | https://istio.io/latest/blog/2022/cryptomb-privatekeyprovider/ |
| Service Mesh – Istio and Envoy Optimizations for Intel® Xeon® Scalable Processors Solution Brief | https://networkbuilders.intel.com/solutionslibrary/service-mesh-istio-envoy-optimizations-intel-xeon-sp-solution-brief |
| 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 - 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 |
| 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 |

# 2 Design

## 2.1 Intel® QuickAssist Technology

4th Gen Intel Xeon Scalable processor has 4xxx-series Intel QAT devices included on the Silicon. Support for the 4xxx Intel QAT devices is included in the mainline kernel and Intel® qatlib library (https://github.com/intel/qatlib). The support for finding and exposing Intel QAT devices to Envoy containers in kubernetes is provided by the Intel QAT device plugin (https://github.com/intel/intel-device-plugins-for-kubernetes/blob/main/cmd/qat_plugin/README.md).

Any earlier Intel QAT devices (such as those which may be present in previous generations of Intel® Xeon® Scalable processor) are not supported by qatlib. Since qatlib does not recognize those devices, they cannot be used by Envoy.

By default, the handshakes in Envoy are synchronous, that is, the handshake function blocks the Envoy worker thread execution until the handshake has been completed. This will not work in a scheme such as Intel QAT acceleration, because the Intel QAT performance benefit comes from the fact that Envoy is ready to do more processing while the QAT device handles the cryptographic operations in parallel. If the Intel QAT calls were synchronous, there will not be any performance benefit. To facilitate asynchronous processing, Envoy has an extension type called ”private key provider”, which performs the following two functions:

- Allows running custom code for private key sign and decrypt operations
- Allows asynchronous handshakes

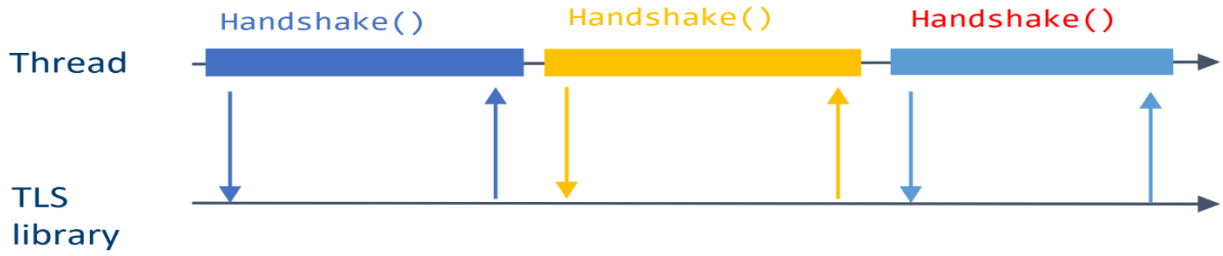


Figure 1. Synchronous TLS handshakes; Handshake function blocks until handshake is completed.

Hence, the Intel QAT private key provider was implemented to allow for Intel QAT handshake processing. When the private key provider is loaded, the handshake function call returns immediately and a callback is evoked when the handshake is ready to be completed, that is, the cryptographic operation is ready.

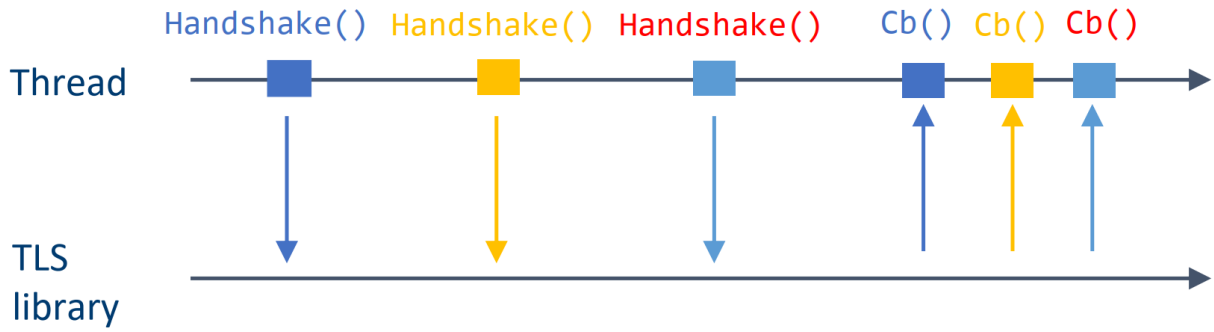


Figure 2. Asynchronous TLS handshakes; TLS library returns immediately during handshake function and a callback is invoked after the handshake is completed.

Internally, the Intel QAT private key provider first queries the hardware. Based on this, it sets up load balancing between the Intel QAT instances that it can find. The load balancing uses a round-robin principle. The cryptographic operations are evenly distributed between the Intel QAT instances and every Intel QAT instance has a corresponding polling thread set up to poll operation completion from the Intel QAT instance.

When the private key provider receives a cryptographic request, it first transforms it to fit the Intel QAT internal data structures and submits it to an instance for processing. The polling thread is then notified to start querying the instance periodically. When the Intel QAT endpoint is ready, an internal callback function is called, which then notifies the worker thread over an internal IPC mechanism. The worker thread asks the upper layer to redo the handshake, which instantly returns with the calculated cryptographic value.

The Intel QAT private key provider polling threads have one parameter, which is user configurable that has a performance trade-off: the poll delay.

Having a small value there improves latency because the polling thread notices that the processing is complete sooner. However, a small value might lead to huge CPU utilization because the polling thread runs too often, and throughput may decrease.

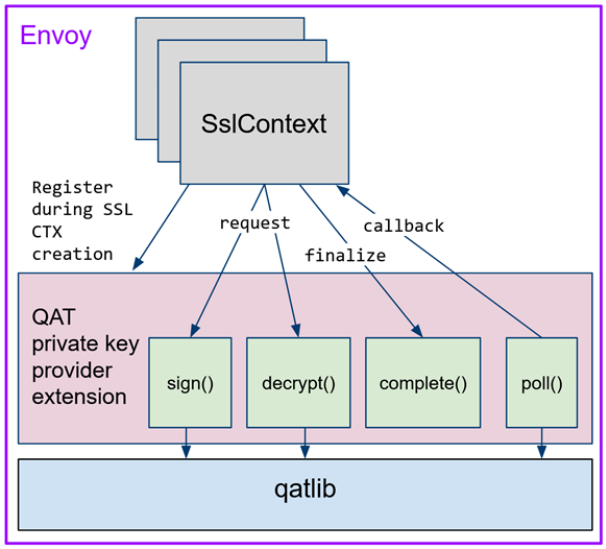


Figure 3. Envoy with Intel QATLib

## 2.2 Intel® Advanced Vector Extensions 512

### 2.2.1 Asymmetric cryptography

The CryptoMb private key provider uses Intel AVX-512 multi-buffer instructions for accelerating handshakes. The Intel AVX-512 instructions are present starting with the 3rd Gen Intel Xeon Scalable processors, and they do not require any special hardware enabling. Running Envoy on a suitable platform and enabling the CryptoMb private key provider in the Envoy configuration is sufficient. The multi-buffer instructions gather several RSA operations into a shared buffer. When the 8-slot buffer is full or when a timer expires, the RSA operations are processed using SIMD (single instruction, multiple data) instructions, which provide greater throughput than processing the RSA operations separately. The downside of this approach is the potentially increased latency because operations may need to wait in the buffer before the processing can be done.

Intel® Integrated Performance Primitives Cryptography (Intel® IPP Cryptography) has a sub-library CryptoMB for cryptographic operation acceleration. See https://github.com/intel/ipp-crypto/tree/develop/sources/ippcp/crypto_mb for the source code. The MultiBuffer instructions operate on a buffer of eight RSA operations. The RSA operations gathered in the buffer are then processed simultaneously.

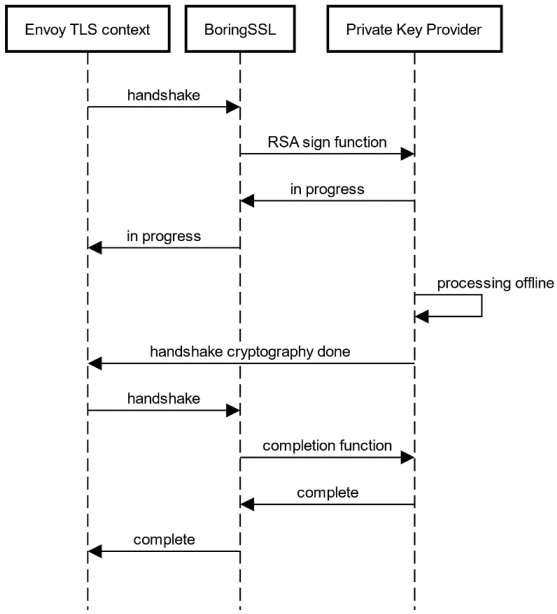


Figure 4. Private key provider handshake flow

By default, the handshakes in Envoy are synchronous, that is, the handshake function blocks the Envoy worker thread execution until the handshake is completed. This will not work in a scheme such as Intel AVX-512 MultiBuffer acceleration, where the Envoy worker thread is supposed to add more operations to the queue. To facilitate this, Envoy has an extension type called "private key providers", which performs the following two functions:

- Allows running custom code for private key sign and decrypt operations
- Allows asynchronous handshakes

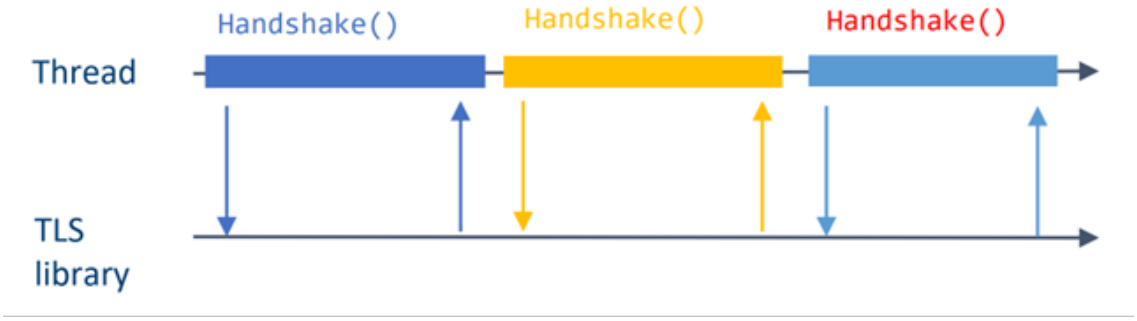


Figure 5. Synchronous TLS handshakes; Handshake function blocks until handshake is completed.

Hence, the CryptoMb private key provider was implemented to allow for Intel AVX-512 handshake processing. When the private key provider is loaded, the handshake function call returns immediately and a call-back is evoked when the handshake is ready to be completed, meaning that the cryptographic operation is ready.

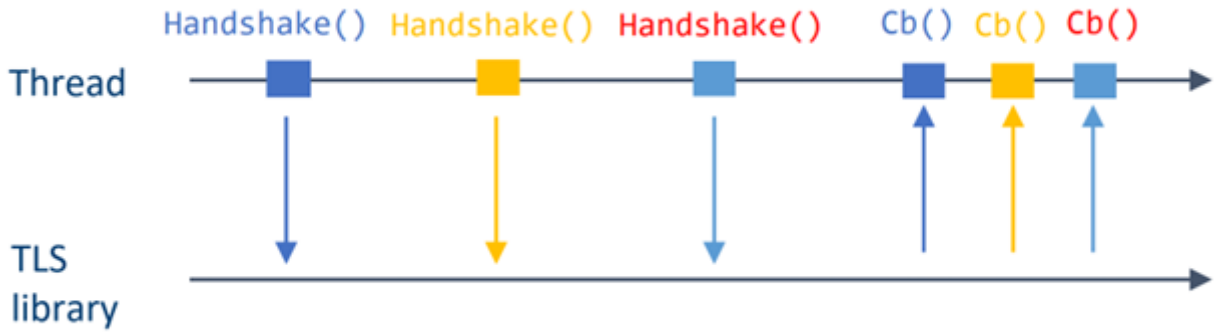


Figure 6. Asynchronous TLS handshakes; TLS library returns immediately during handshake function and a call-back is invoked after the handshake is completed.

There is an 8-slot RSA operations buffer for each Envoy worker thread. The number of Envoy worker threads normally corresponds to the number of CPU cores on which Envoy is executing. For example, in a system with ten worker threads, there are in total 80 "slots" for RSA operations, and Envoy fairly distributes incoming connections between the worker threads. This means that if there are many worker threads and if connections come in on a slower rate, the system will see increased latencies because the buffers fill up slowly.

To keep the latencies tolerable, the CryptoMb private key provider needs to be configured with a poll delay. The poll delay defines the duration of the timer that triggers the processing of CryptoMb in case the buffers haven't filled up before the timer expires. The timer is started when the first handshake in the queue is received and reset during the processing of the queue. If the queue size hits the maximum value (eight queued operations) the SIMD operation can be executed without waiting for the timer to trigger. Shorter poll delay values lead to shorter latencies at low loads because the timer triggers more quickly for partially filled queues. Longer poll delay values lead to more simultaneously handled handshakes (if the load is sufficient). Determining a good poll delay value can be challenging but according to testing done so far, a 10 ms poll delay value can work well as a compromise between good performance at high loads and short latencies at low loads. Changing the poll delay value from 10 ms might lead to better or worse performance but it is a good idea to keep it in a similar range (i.e., a short duration of milliseconds).

CryptoMB has an Envoy histogram statistic to help with seeing the frequency of processed queue sizes. The more occurrences of larger queue sizes can be seen in the statistics, the more efficiently CryptoMB is working by processing more operations simultaneously. However, the processed queue sizes depend heavily on the incoming load and no conclusions be made by observing only the queue size statistics.

### 2.2.2 Symmetric cryptography

In addition to public key cryptography, AES symmetric cryptography can also be accelerated. Intel® Managed Distribution of Istio* Service Mesh have support for vector AES (vAES) multi-buffer operations up until release 22.03. The feature is removed from release 22.06 onwards.

The vAES patch set adds transparent acceleration to BoringSSL using Intel AVX-512 instructions. The acceleration is transparent in the sense that no configuration is needed. If the processor supports the necessary Intel AVX-512 operations, the speedup is automatic. The performance improvement depends on the processor model, the key size, and the buffer length (amount of data to be encrypted at once).

Table 3. Speedup of vAES (times the baseline performance) on Intel® Xeon® 8380 Platinum @ 2.30GHz

| | Payload in bytes | | | | |
|---|---|---|---|---|---|
| | 16 | 256 | 1350 | 8192 | 16384 |
| AES-128-GCM | 1.08 | 1.15 | 1.31 | 1.95 | 2.07 |
| AES-256-GCM | 1.08 | 1.14 | 1.29 | 2.05 | 2.14 |

No hyperthreading, measured by `bssl speed`. Smallest performance increase is on short payload lengths, and the biggest performance increase is on 16k buffers with a 256-bit AES key.

The vAES BoringSSL patch set is available from https://boringssl-review.googlesource.com/c/boringssl/+/48745. The patch set is not merged in mainline BoringSSL and it is not expected to be in the near future.

# 3 Deployment

## 3.1 Intel® QAT Deployment

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.

### 3.1.1 Hardware Setup

Note that to enable the QAT private key provider, you must have suitable hardware on which the Envoy proxy is running. The hardware needs to be set up roughly as follows (this may require changes depending on your Linux kernel version and distribution):

1. Make sure that VT-d is enabled in BIOS to get IOMMU support. Make sure that IOMMU is enabled in the kernel command line.
2. Make sure that you have QAT firmware installed from the Linux Firmware repository (https://git.kernel.org/pub/scm/linux/kernel/git/firmware/linux-firmware.git). Make sure QAT_4XXX driver is enabled in your kernel as a module. The driver has been in the kernel since 5.11 but tested only on 5.17 onwards. Load the driver (`modprobe qat_4xxx`).
3. Load the vfio-pci driver (`modprobe vfio-pci`).
4. Choose a QAT physical endpoint, which has cryptography support. The in-tree driver has half of the physical endpoints reserved for cryptography and half for compression. This will become configurable in future kernel versions. You can check the features from debugfs (`grep ServicesEnabled /sys/kernel/debug/qat_4xxx_0000\:6b\:00.0/dev_cfg` for endpoint qat_4xxx_0000:6b:00.0). If you have `ServicesEnabled = dc` it means that the physical endpoint is configured for compression, and if you have `ServicesEnabled = cy` it means that the endpoint is configured for cryptography. If in doubt, enable all endpoints.
5. Create SRIOV instances out of the physical endpoint. You can echo 16 to the physical endpoint's sriov_numfs file (`echo 16 > /sys/devices/pci0000\:6b/0000\:6b\:00.0/sriov_numvfs`).
6. Bind suitable sriov_numfs devices to the vfio-pci driver. You can either do this manually or use the convenient dpdk-devbind.py script (`python3 /home/ipuustin/dpdk-devbind.py -b vfio-pci 0000:6b:01.4` and so on for all the SRIOV devices). The script is available from https://github.com/DPDK/dpdk/blob/main/usertools/dpdk-

devbind.py. You can alternatively load the vfio-pci driver in such a way that it binds the QAT VFs directly (`modprobe vfio-pci ids=8086:4941`). In case the vfio-pci driver is built into the kernel, you can specify this on the kernel command line (vfio-pci.ids).

7. After this is performed, you should see a bunch of devices in `/dev/vfio/`. Optional: you can change their permissions in such a way that the Envoy process running in container can use them (`chmod a+rw /dev/vfio/*`). If you do not do this, you need to tune your container runtime's device node re-creation parameters as explained below.

After the hardware is set up, it is recommended to test whether QAT works at this point.

### 3.1.2 Container Environment Setup

After the hardware starts working, set up the container environment. You often need to increase the amount of lockable IPC memory, which can be granted to the containers. For example, if you are using containerd, add this file as `/etc/systemd/system/containerd.service.d/memlock.conf`:

```
[Service]
LimitMEMLOCK=16777216
```

You can also set up containerd in such a way that the QAT device recreation happens with suitable security context (to allow non-root users access to the VFIO device files). You will need to set RunAsUser or RunAsGroup in your pod's security context and add this to the containerd configuration:

```
[plugins."io.containerd.grpc.v1.cri"]
    device_ownership_from_security_context = true
```

Then, restart containerd: `systemctl daemon-reload && systemctl restart containerd`

If the Envoy proxy is running on a host, which does not have Intel QAT cryptography VF support, any attempt to enable the Intel QAT private key provider (either during startup or dynamically over SDS (Secret Discovery Service)) will fail.

### 3.1.3 Kubernetes

The Kubernetes device plugin can be used to find QAT resources and setup VFIO devices inside the container. Read the Intel QAT device plugin documentation here https://github.com/intel/intel-device-plugins-for-kubernetes/blob/main/cmd/qat_plugin/README.md. Note that you can skip the device binding step in hardware setup if you do this. The Intel QAT device plugin will bind the Intel QAT VFs properly to the vfio-pci driver.

When you add Intel QAT resources to the Envoy/Istio pods, make sure that you are requesting the crypto resource.

```
resources:
 requests:
  qat.intel.com/cy: '1'
```

Note that if you increase the number of resources, which are allocated to the container you can potentially get more performance, because the Intel QAT private key provider will automatically set up load balancing between the available Intel QAT instances. If the crypto VFIO devices are made from different physical Intel QAT endpoints there is a speedup potential. But, if the VFIO devices come from the same physical Intel QAT endpoint, there will be no performance increase.

### 3.1.4 Istio

The latest Istio yaml-file can be found here: https://intel.github.io/istio/README.html.

### 3.1.5 Envoy

To test Envoy Intel QAT, perform the following steps on a 4th Gen Intel Xeon scalable processor:

1. Generate a key-certificate pair. Depending on your setup, you may also need to run `chmod a+r key.pem` to the resulting key file (due to potentially different Docker user UIDs).

```
$ openssl req -x509 -new -batch -newkey rsa:2048 -nodes -subj '/CN=localhost' -keyout key.pem -out cert.pem
```

2. Create an Envoy configuration file. The configuration below returns HTTP code “200” for every incoming request. Save it to the name “conf.yaml”.

```
static_resources:
  listeners:
  - address:
      socket_address:
        address: 0.0.0.0
        port_value: 9000
    filter_chains:
      transport_socket:
        name: envoy.transport_sockets.tls
        typed_config:
          "@type":
type.googleapis.com/envoy.extensions.transport_sockets.tls.v3.DownstreamTlsContext
          common_tls_context:
            tls_certificates:
              certificate_chain: { "filename": "/etc/ssl/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: 0.005s
                  private_key: { "filename": "/etc/ssl/key.pem" }
      filters:
      - name: envoy.http_connection_manager
        typed_config:
          "@type":
type.googleapis.com/envoy.extensions.filters.network.http_connection_manager.v3.HttpConnect
ionManager
          codec_type: auto
          stat_prefix: ingress_http
          route_config:
            name: local_route
            virtual_hosts:
            - name: backend
              domains:
              - "*"
              routes:
              - match: { prefix: / }
                direct_response: { status: 200 }
          http_filters:
          - name: envoy.filters.http.router
            typed_config: {}
admin:
  access_log_path: "/dev/null"
  address:
    socket_address:
      address: 0.0.0.0
      port_value: 9001
```

3. Start Envoy Istio image and mount the configuration file, key, and certificate to the container. This command also exposes port 9000 from the container to the host. You need to add to the command line –device=/dev/vfio/<number>, where <number> is a QAT crypto VFIO instance.

```
$ docker run -ti --rm -p 9000:9000 –device=/dev/vfio/vfio -v
$(pwd)/conf.yaml:/tmp/conf.yaml -v $(pwd)/cert.pem:/etc/ssl/cert.pem -v
$(pwd)/key.pem:/etc/ssl/key.pem --entrypoint=envoy intel/proxyv2:latest -c
/tmp/conf.yaml
```

4. From outside of the container, access Envoy port 9000 with curl to verify that everything works:

```
$ curl --cacert $(pwd)/cert.pem https://localhost:9000 -v
```

## 3.2 Intel® AVX-512 Deployment

Note that to enable the CryptoMb private key provider, you must have suitable hardware on which the Envoy proxy is running. If the Envoy proxy is running on a host, which does not have Intel AVX-512 support, any attempt to enable the CryptoMb private key provider (either during startup or dynamically over SDS) will fail.

### 3.2.1 Istio

The latest Istio yaml-file for cryptomb can be found here. A test setup using k6, Istio, and a Fortio server.

Below are instructions to view the queue size statistics when using Istio. When using the commands, the istio-ingressgateway pod name (istio-ingressgateway-7c86d77d76-qrfbx) must be replaced with the name of the pod used in your setup.

The following command can be used to view the queue size statistics of cryptomb using the prometheus format (works since release 22.03):

```
kubectl exec istio-ingressgateway-7c86d77d76-qrfbx -n istio-system -c istio-proxy -- pilot-agent request GET "stats?format=prometheus&filter=cryptomb"

Example output:

# TYPE envoy_listener_cryptomb_rsa_queue_sizes histogram
envoy_listener_cryptomb_rsa_queue_sizes_bucket{listener_address="0.0.0.0_8443",le="2"} 178
envoy_listener_cryptomb_rsa_queue_sizes_bucket{listener_address="0.0.0.0_8443",le="3"} 379
envoy_listener_cryptomb_rsa_queue_sizes_bucket{listener_address="0.0.0.0_8443",le="4"} 567
envoy_listener_cryptomb_rsa_queue_sizes_bucket{listener_address="0.0.0.0_8443",le="5"} 791
envoy_listener_cryptomb_rsa_queue_sizes_bucket{listener_address="0.0.0.0_8443",le="6"} 1018
envoy_listener_cryptomb_rsa_queue_sizes_bucket{listener_address="0.0.0.0_8443",le="7"} 1256
envoy_listener_cryptomb_rsa_queue_sizes_bucket{listener_address="0.0.0.0_8443",le="8"} 1550
envoy_listener_cryptomb_rsa_queue_sizes_bucket{listener_address="0.0.0.0_8443",le="9"} 13081
envoy_listener_cryptomb_rsa_queue_sizes_bucket{listener_address="0.0.0.0_8443",le="+Inf"} 13081
envoy_listener_cryptomb_rsa_queue_sizes_sum{listener_address="0.0.0.0_8443"} 99563.050000000002910383045673337
envoy_listener_cryptomb_rsa_queue_sizes_count{listener_address="0.0.0.0_8443"} 13081
```

The output contains **cumulative** buckets, which have upper bounds, e.g., le="2" has an upper bound of 2 (le stands for less than or equal to but because of a bug it is only less than). All values in buckets with smaller upper bounds are included in buckets with higher upper bounds (e.g., the bucket with le="2" always has a value <= le="3"). The le="2" bucket value tells the frequency of when the queue size is equal to 1. The le="3" bucket value tells the frequency of when the queue size was equal to 1 or 2. By subtracting the value of bucket le="2" from le="3" the frequency of queue size 2 can be deducted and so on.

The following command can be used to view the queue size statistics of cryptomb using an Envoy statistics format (does not work in release 22.03 but should work in a later release):

```
kubectl exec istio-ingressgateway-7c86d77d76-qrfbx -n istio-system -c istio-proxy -- pilot-agent request GET "stats?histogram_buckets=disjoint&filter=cryptomb"

Example output:

listener.0.0.0.0_9000.cryptomb.rsa_queue_sizes: B2(0,1) B3(0,0) B4(0,0) B5(0,0) B6(0,0) B7(0,0) B8(0,0) B9(0,0)
```

The output contains **disjoint** buckets, which have upper bounds (e.g., B2 has an upper bound of 2, the upper bounds work exactly like the Prometheus output explained above). However, since the buckets are disjoint, they don't contain values from other buckets (e.g., B9 would only contain the amount of queue size 8 occurrences).

### 3.2.2 Envoy

An example configuration for running Envoy with Docker can be found here: https://intel.github.io/istio/README.html. The directory contains scripts for running Envoy with different configs. A load generator script can also be found to test the Envoy configuration.

The CryptoMb private key provider has been part of the Envoy contrib image since Envoy version 1.20. In order to test Envoy CryptoMb, you need to perform the following on a 3rd Gen Intel Xeon Scalable processor or later:

1. Generate a key-certificate pair. Depending on your setup, you may also need to run chmod a+r key.pem to the resulting key file (due to potentially different Docker user UIDs).

```
$ openssl req -x509 -new -batch -newkey rsa:2048 -nodes -subj '/CN=localhost' -keyout key.pem -
out cert.pem
```

2. Create an Envoy configuration file. The configuration below just returns HTTP code "200" for every incoming request. Save it with the name "conf.yaml".

```
static_resources:
 listeners:
 - address:
  socket_address:
   address: 0.0.0.0
   port_value: 9000
 filter_chains:
  transport_socket:
   name: envoy.transport_sockets.tls
   typed_config:
    "@type":
type.googleapis.com/envoy.extensions.transport_sockets.tls.v3.DownstreamTlsContext
    common_tls_context:
     tls_certificates:
      certificate_chain: { "filename": "/etc/ssl/cert.pem" }
      private_key_provider:
       provider_name: cryptomb
       typed_config:
        "@type":
"type.googleapis.com/envoy.extensions.private_key_providers.cryptomb.v3alpha.CryptoMbPrivate
KeyMethodConfig"
       poll_delay: 0.01s
       private_key: { "filename": "/etc/ssl/key.pem" }
  filters:
  - name: envoy.http_connection_manager
   typed_config:
    "@type":
type.googleapis.com/envoy.extensions.filters.network.http_connection_manager.v3.HttpConnecti
onManager
    codec_type: auto
    stat_prefix: ingress_http
    route_config:
     name: local_route
     virtual_hosts:
     - name: backend
      domains:
      - "*"
      routes:
      - match: { prefix: / }
       direct_response: { status: 200 }
    http_filters:
```

```
    - name: envoy.filters.http.router
     typed_config: {}
admin:
 access_log_path: "/dev/null"
 address:
  socket_address:
   address: 0.0.0.0
   port_value: 9001
```

3. Start the Envoy contrib image and mount the configuration file, key, and certificate to the container. This command also exposes port 9000 from the container to the host.

```
$ docker run -ti --rm -p 9000:9000 -v $(pwd)/conf.yaml:/tmp/conf.yaml -v
$(pwd)/cert.pem:/etc/ssl/cert.pem -v $(pwd)/key.pem:/etc/ssl/key.pem envoyproxy/envoy-
contrib:v1.21.2 -c /tmp/conf.yaml
```

4. From outside of the container, access Envoy port 9000 with curl to verify that everything works:

```
$ curl --cacert $(pwd)/cert.pem https://localhost:9000 -v
```


Figure 7. Intel® Distribution of Istio Grafana Dashboard

## 4 Summary

Service mesh deployments that use Istio with Envoy cause latency and performance challenges due to the nature of the 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). The performance benefits on 4th Gen Intel® Xeon® Scalable processor 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 Intel QAT, for 8 core and 16 cores can save 42% and 60% respectively on the 4th Gen Intel Xeon Scalable processor CPU cycles
- Using 2x Intel QAT, for 8 core and 16 cores can save 18% and 49% for 2x QAT on the 4th Gen Intel Xeon Scalable processor CPU cycles

More on the performance charts are described in the “Service Mesh – Istio and Envoy Optimizations for Intel® Xeon® Scalable Processors” Solution Brief.

## Appendix 1: Debugging ISTIO QAT Issues

Common errors and solutions:

1. Envoy does not start. Check dmesg and console.
    a. If the device file cannot be found, probably there is no resource assigned to the container.
    b. If the device file cannot be opened, the container probably does not have read and write access to it.
    c. If there are memory errors, check that memlock ulimit in the container is high enough. Also try adding IPC_LOCK capability. Alternatively IOMMU support may be missing.
    d. If there are no errors in dmesg, probably the right Intel QAT instance type is missing (dc instead of cy).
2. The default value for the maximum number of devices in Kubernetes QAT device plugin is 16. If you want to have more devices managed by Kubernetes, add the argument "-max-num-devices=xxx" in the daemonset configuration.
3. If you are using Ubuntu with the apparmor policy set, containers may not be allowed to open QAT devices. You can use this kustomization.yaml file to deploy the Intel QAT device plugin.

Sometimes you do not see any performance benefit with the Intel QAT private key provider. Please check the following to verify that your setup is correct:

1. Are you using an RSA key? Intel QAT acceleration is enabled only for RSA keys (for now).
2. Try changing the poll delay. 5ms is a good starting point.
3. Do you get reasonable performance numbers in your benchmarking without acceleration? Envoy can handle without acceleration (very roughly) 500-1500 RSA 2k handshakes per core. If you are seeing less, you probably have a bottleneck elsewhere in your benchmarking setup. If you are seeing much more, it's likely that your test setup is in fact reusing SSL session keys and the handshakes are not performing new RSA operations at all. Many HTTPs load generators have an option to configure this. For the same reason you can't expect much performance increase from sidecar TLS acceleration, because TLS handshakes are not very common there.
4. Is the Envoy listening socket really configured to use Intel QAT? You can dump the configuration with these commands:
```
kubectl port-forward -n istio-system istio-ingressgateway-change-me 15000 &
curl localhost:15000/config_dump
```
5. Are you running Envoy on an isolated CPUset? Envoy performance is sensitive to cache issues, and the best performance is had when the CPUset is separate from other payloads in the system. You can use the Kubernetes CPU manager static policy to achieve this.
6. Make sure that the number of Envoy worker threads is equal to the CPUset size. You can use Envoy's `--cpuset-threads` parameter or set the `concurrency` parameter manually.
7. Read the official Envoy benchmarking checklist and apply the steps (as applicable): https://www.envoyproxy.io/docs/envoy/latest/faq/performance/how_to_benchmark_envoy

## Appendix 2: Debugging Istio CryptoMB Issues

Sometimes you do not see any performance benefit with the CryptoMb private key provider. Please check the following to verify that your setup is correct:

1. Are you using an RSA key? CryptoMb acceleration is enabled only for RSA keys. ECDSA support is disabled due to BoringSSL internal restrictions.
2. Do you get reasonable performance numbers in your benchmarking without acceleration? Envoy can handle without acceleration (very roughly) 500-1500 RSA 2k handshakes per core. If you are seeing less, you probably have a bottleneck elsewhere in your benchmarking setup. If you are seeing much more, it's likely that your test setup is in fact reusing SSL session keys and the handshakes are not performing new RSA operations at all. Many HTTPs load generators have an option to configure this.
3. Is the Envoy listening socket really configured to use CryptoMb? You can dump the configuration with these commands:
```
kubectl port-forward -n istio-system istio-ingressgateway-change-me 15000 &
curl localhost:15000/config_dump
```
4. Are you running Envoy on an isolated CPUset? Envoy performance is sensitive to cache issues, and the best performance is had when the CPUset is separate from other payloads in the system. You can use Kubernetes CPU manager static policy to achieve this.
5. Make sure that the number of Envoy worker threads is equal to the CPUset size. You can use the Envoy's cpuset-threads parameter or set the concurrency parameter manually.

Read the official Envoy benchmarking checklist and apply the steps there (as applicable):
https://www.envoyproxy.io/docs/envoy/latest/faq/performance/how_to_benchmark_envoy



0123/DN/WIT/PDF 763503-001US