# Table of contents:

1 **Introduction to Cilium**
   - 1.1 What is Cilium? .................................................. 1
   - 1.2 Why Cilium? ......................................................... 1
   - 1.3 Documentation Roadmap ........................................... 2
   - 1.4 Getting Help ......................................................... 2

2 **Getting Started Guide**
   - 2.1 Step 0: Install Vagrant ........................................... 3
   - 2.2 Step 1: Download the Cilium Source Code ......................... 3
   - 2.3 Step 2: Starting the Docker + Cilium VM .......................... 4
   - 2.4 Step 3: Accessing the VM ......................................... 4
   - 2.5 Step 4: Confirm that Cilium is Running .......................... 4
   - 2.6 Step 5: Create a Docker Network of Type Cilium ................. 5
   - 2.7 Step 6: Start an Example Service with Docker .................. 5
   - 2.8 Step 7: Apply an L3/L4 Policy With Cilium ...................... 5
   - 2.9 Step 8: Test L3/L4 Policy ......................................... 6
   - 2.10 Step 9: Apply and Test an L7 Policy with Cilium ................ 6
   - 2.11 Step 10: Clean-Up .................................................. 8

3 **Architecture Guide**
   - 3.1 Cilium Components ................................................. 9
   - 3.2 Address Management ............................................... 11
   - 3.3 IP Interconnectivity ............................................... 13
   - 3.4 Security ............................................................ 14
   - 3.5 Integration with Container Platforms ................................ 22

4 **Administrator Guide**
   - 4.1 Vagrant ............................................................. 24
   - 4.2 Prerequisites - Linux Kernel Version ............................. 25
   - 4.3 Installing Cilium .................................................... 25
   - 4.4 Container Node Network Configuration ............................ 28
   - 4.5 Configuring Cilium to use a Key-Value Store ...................... 30
   - 4.6 Container Platform Integrations ................................... 30
   - 4.7 Cilium Agent Command Line Options .............................. 32
   - 4.8 Cilium CLI Commands .............................................. 33
   - 4.9 Troubleshooting ...................................................... 33
### 5 BPF and XDP Reference Guide

- 5.1 BPF Architecture ................................................................. 34
- 5.2 Toolchain ......................................................................... 42
- 5.3 tc (traffic control) ................................................................. 61
- 5.4 XDP ................................................................................. 61
- 5.5 References ......................................................................... 62

### 6 API Reference ....................................................................... 63

### 7 Developer / Contributor Guide ............................................. 71

- 7.1 Setting up the a development environment ............................. 71
- 7.2 Testsuite ............................................................................. 71
- 7.3 Submitting a pull request ...................................................... 72
- 7.4 Release Process ................................................................ 72
- 7.5 Developer’s Certificate of Origin ........................................ 72

**HTTP Routing Table** .................................................................. 81
CHAPTER 1

Introduction to Cilium

What is Cilium?

Cilium is open source software for transparently securing the network connectivity between application services deployed using Linux container management platforms like Docker and Kubernetes.

At the foundation of Cilium is a new Linux kernel technology called BPF, which enables the dynamic insertion of powerful security visibility and control logic within Linux itself. Because BPF runs inside the Linux kernel itself, Cilium security policies can be applied and updated without any changes to the application code or container configuration.

Why Cilium?

The development of modern datacenter applications has shifted to a service-oriented architecture often referred to as microservices, wherein a large application is split into small independent services that communicate with each other via APIs using lightweight protocols like HTTP. Microservices applications tend to be highly dynamic, with individual containers getting started or destroyed as the application scales out / in to adapt to load changes and during rolling updates that are deployed as part of continuous delivery.

This shift toward highly dynamic microservices presents both a challenge and an opportunity in terms of securing connectivity between microservices. Traditional Linux network security approaches (e.g., iptables) filter on IP address and TCP/UDP ports, but IP addresses frequently churn in dynamic microservices environments. The highly volatile life cycle of containers causes these approaches to struggle to scale side by side with the application as load balancing tables and access control lists carrying hundreds of thousands of rules that need to be updated with a continuously growing frequency. Protocol ports (e.g. TCP port 80 for HTTP traffic) can no longer be used to differentiate between application traffic for security purposes as the port is utilized for a wide range of messages across services.

An additional challenge is the ability to provide accurate visibility as traditional systems are using IP addresses as primary identification vehicle which may have a drastically reduced lifetime of just a few seconds in microservices architectures.

By leveraging Linux BPF, Cilium retains the ability to transparently insert security visibility + enforcement, but does so in a way that is based on service / pod / container identity (in contrast to IP address identification in traditional systems) and can filter on application-layer (e.g. HTTP). As a result, Cilium not only makes it simple to apply security policies
in a highly dynamic environment by decoupling security from addressing, but can also provide stronger security
isolation by operating at the HTTP-layer in addition to providing traditional Layer 3 and Layer 4 segmentation.
The use of BPF enables Cilium to achieve all of this in a way that is highly scalable even for large-scale environments.

Documentation Roadmap

The remainder of this documentation is divided into four sections:

- **Getting Started Guide**: Provides a simple tutorial for running a small Cilium setup on your laptop. Intended as
  an easy way to get your hands dirty applying Cilium security policies between containers.
- **Architecture Guide**: Describes the components of the Cilium architecture, and the different models for deploy-
ing Cilium. Provides the high-level understanding required to run a full Cilium deployment and understand its
  behavior.
- **Administrator Guide**: Details instructions for installing, configuring, and troubleshooting Cilium in different
  deployment modes.
- **Developer / Contributor Guide**: Give background to those looking to develop and contribute modifications to
  the Cilium code or documentation.

Getting Help

We use Github issues to maintain a list of Cilium Frequently Asked Questions (FAQ) . Check there to see if your
question(s) is already addressed.

The best way to get help if you get stuck is to contact us on the Cilium Slack channel.

If you are confident that you have found a bug, or if you have a feature request, please go ahead and create an issue on
our Github.

If you are interested in contributing to the code or docs, ping us on Slack or just dive in on Github!
This document serves as the easiest introduction to Cilium. It is a detailed walk through of getting a single-node Cilium + Docker environment running on your laptop. It is designed to take 15-30 minutes.

If you haven’t read the Introduction to Cilium yet, we’d encourage you to do that first.

The tutorial leverages Vagrant, and as such should run on any operating system supported by Vagrant, including Linux, MacOS X, and Windows. The VM running Docker + Cilium requires about 3 GB of RAM, so if your laptop has limited resources, you may want to close other memory intensive applications.

The vagrant box is currently available for the following hypervisors. Please contact us on slack to request building for additional hypervisors.

- VirtualBox
- libvirt

The best way to get help if you get stuck is to ask a question on the Cilium Slack channel. With Cilium contributors across the globe, there is almost always someone available to help.

**Step 0: Install Vagrant**

*Note:* You need to run Vagrant version 1.8.3 or later or you will run into issues booting the Ubuntu 16.10 base image.

If you don’t already have Vagrant installed, follow the Vagrant Install Instructions

**Step 1: Download the Cilium Source Code**

Download the latest Cilium source code and unzip the files.

Alternatively, if you are a developer, feel free to use Git to clone the repository:
Step 2: Starting the Docker + Cilium VM

Open a terminal and navigate into the top of the cilium source directory.

Then navigate into `examples/getting-started` and run `vagrant up`:

```bash
$ cd examples/getting-started
$ vagrant up
```

The script usually takes a few minutes depending on the speed of your internet connection. Vagrant will set up a VM, install the Docker container runtime and run Cilium with the help of Docker compose. When the script completes successfully, it will print:

```bash
==> cilium-1: Creating cilium-kvstore
==> cilium-1: Creating cilium
==> cilium-1: Creating cilium-docker-plugin
```

If the script exits with an error message, do not attempt to proceed with the tutorial, as later steps will not work properly. Instead, contact us on the Cilium Slack channel.

Step 3: Accessing the VM

After the script has successfully completed, you can log into the VM using `vagrant ssh`:

```bash
$ vagrant ssh
```

All commands for the rest of the tutorial below should be run from inside this Vagrant VM. If you end up disconnecting from this VM, you can always reconnect in a new terminal window just by running `vagrant ssh` again from the Cilium directory.

Step 4: Confirm that Cilium is Running

The Cilium agent is now running as a system service and you can interact with it using the `cilium` CLI client. Check the status of the agent by running `cilium status`:

```bash
$ cilium status
KVStore: Ok
ContainerRuntime: Ok
Kubernetes: Disabled
Cilium: Ok
```

The status indicates that all components are operational with the Kubernetes integration currently being disabled.
Step 5: Create a Docker Network of Type Cilium

Cilium integrates with local container runtimes, which in the case of this demo means Docker. With Docker, native networking is handled via a component called libnetwork. In order to steer Docker to request networking of a container from Cilium, a container must be started with a network of driver type “cilium”.

With Cilium, all containers are connected to a single logical network, with isolation added not based on IP addresses but based on container labels (as we will do in the steps below). So with Docker, we simply create a single network named ‘cilium-net’ for all containers:

```sh
docker network create --ipv6 --subnet ::1/112 --driver cilium --ipam-driver cilium
```

Step 6: Start an Example Service with Docker

In this tutorial, we’ll use a container running a simple HTTP server to represent a microservice which we will refer to as Service1. As a result, we will start this container with the label “id.service1”, so we can create Cilium security policies for that service.

Use the following command to start the Service1 container connected to the Docker network managed by Cilium:

```sh
docker run -d --name service1-instance1 --net cilium-net -l "id.service1" cilium/demo-httpd e5723edaa2a1307e7aa7e71b4087882de0250973331bc74a37f6f80667bc5856
```

This has launched a container running an HTTP server which Cilium is now managing as an endpoint. A Cilium endpoint is one or more application containers which can be addressed by an individual IP address.

Step 7: Apply an L3/L4 Policy With Cilium

When using Cilium, endpoint IP addresses are irrelevant when defining security policies. Instead, you can use the labels assigned to the VM to define security policies, which are automatically applied to any container with that label, no matter where or when it is run within a container cluster.

We’ll start with an overly simple example where we create two additional services, Service2 and Service3, and we want Service2 containers to be able to reach Service1 containers, but Service3 containers should not be allowed to reach Service1 containers. Additionally, we only want to allow Service1 to be reachable on port 80, but no other ports. This is a simple policy that filters only on IP address (network layer 3) and TCP port (network layer 4), so it is often referred to as an L3/L4 network security policy.

Cilium performs stateful ‘connection tracking’, meaning that if policy allows the Service2 to contact Service3, it will automatically allow return packets that are part of Service1 replying to Service2 within the context of the same TCP/UDP connection.

We can achieve that with the following Cilium policy:

```json
{
    "name": "root",
    "rules": [
        {
            "coverage": ["id.service1"],
            "allow": ["id.service2"]
        },
        {
            "coverage": ["id.service1"],
            "allow": ["id.service3"],
            "reject": ["id.service4"]
        }
    ]
}
```
Save this JSON to a file named `l3_l4_policy.json` in your VM, and apply the policy by running:

```
$ cilium policy import l3_l4_policy.json
```

### Step 8: Test L3/L4 Policy

You can now launch additional containers that represent other services attempting to access `Service1`. Any new container with label “id.service2” will be allowed to access `Service1` on port 80, otherwise the network request will be dropped.

To test this out, we’ll make an HTTP request to `Service1` from a container with the label “id.service2”:

```
$ docker run --rm -ti --net cilium-net -l "id.service2" --cap-add NET_ADMIN cilium/demo-client ping service1-instance1
```

We can see that this request was successful, as we get a valid ping responses.

Now let’s run the same ping request to `Service1` from a container that has label “id.service3”:

```
$ docker run --rm -ti --net cilium-net -l "id.service3" --cap-add NET_ADMIN cilium/demo-client ping service1-instance1
```

You will see no ping replies, as all requests are dropped by the Cilium security policy.

So with this we see Cilium’s ability to segment containers based purely on a container-level identity label. This means that the end user can apply security policies without knowing anything about the IP address of the container or requiring some complex mechanism to ensure that containers of a particular service are assigned an IP address in a particular range.

### Step 9: Apply and Test an L7 Policy with Cilium

In the simple scenario above, it was sufficient to either give `Service2` / `Service3` full access to `Service1`’s API or no access at all. But to provide the strongest security (i.e., enforce least-privilege isolation) between microservices, each service that calls `Service1`’s API should be limited to making only the set of HTTP requests it requires for legitimate operation.

**For example, consider a scenario where `Service1` has two API calls:**

- GET /public
- GET /private
Continuing with the example from above, if `Service2` requires access only to the GET /public API call, the L3/L4 policy along has no visibility into the HTTP requests, and therefore would allow any HTTP request from `Service2` (since all HTTP is over port 80).

To see this, run:

```
$ docker run --rm -ti --net cilium-net -l "id.service2" cilium/demo-client curl -si http://service1-instance1/public
```

```json
{
  "val": "this is public"
}
```

and

```
$ docker run --rm -ti --net cilium-net -l "id.service2" cilium/demo-client curl -si http://service1-instance1/private
```

```json
{
  "val": "this is private"
}
```

Cilium is capable of enforcing HTTP-layer (i.e., L7) policies to limit what URLs `Service2` is allowed to reach. Here is an example policy file that extends our original policy by limiting `Service2` to making only a GET /public API call, but disallowing all other calls (including GET /private).

```json
{
  "name": "root",
  "rules": [{
    "coverage": ["id.service1"],
    "allow": ["id.service2", "reserved:host"]
  },
  {
    "coverage": ["id.service2"],
    "l4": {
      "out-ports": [{
        "port": 80, "protocol": "tcp",
        "l7-parser": "http",
        "l7-rules": [
          { "expr": "Method("GET") && Path("/public")" }
        ]
      }
    }
  }
}
```

Create a file with this contents and name it `l7_aware_policy.json`. Then import this policy to Cilium by running:

```
$ cilium policy import l7_aware_policy.json
```

```
$ docker run --rm -ti --net cilium-net -l "id.service2" cilium/demo-client curl -si http://service1-instance1/public
```

```json
{
  "val": "this is public"
}
```

and

```
$ docker run --rm -ti --net cilium-net -l "id.service2" cilium/demo-client curl -si http://service1-instance1/private
```

Access denied

As you can see, with Cilium L7 security policies, we are able to permit `Service2` to access only the required API resources on `Service1`, thereby implementing a “least privilege” security approach for communication between microservices.

We hope you enjoyed the tutorial. Feel free to play more with the setup, read the rest of the documentation, and feel free to reach out to us on the Cilium Slack channel with any questions!
Step 10: Clean-Up

When you are done with the setup and want to tear-down the Cilium + Docker VM, and destroy all local state (e.g., the VM disk image), open a terminal, navigate to the cilium directory and run:

```bash
$ vagrant destroy cilium-master
```

You can always re-create the VM using the steps described above.

If instead you just want to shut down the VM but may use it later, `vagrant halt cilium-master` will work, and you can start it again later using the contrib/vagrant/start.sh script.
The goal of this document is to describe the components of the Cilium architecture, and the different models for deploying Cilium within your datacenter or cloud environment. It focuses on the higher-level understanding required to run a full Cilium deployment. You can then use the more detailed Administrator Guide to understand the details of setting up Cilium.

**Cilium Components**

A deployment of Cilium consists of the following components running on each Linux container node in the container cluster:
• **Cilium Agent**: Userspace daemon that interacts with the container runtime and orchestration systems such as Kubernetes via Plugins to setup networking and security for containers running on the local server. Provides an API for configuring network security policies, extracting network visibility data, etc.

• **Cilium CLI Client**: Simple CLI client for communicating with the local Cilium Agent, for example, to configure network security or visibility policies.

• **Linux Kernel BPF**: Integrated capability of the Linux kernel to accept compiled bytecode that is run at various hook / trace points within the kernel. Cilium compiles BPF programs and has the kernel run them at key points in the network stack to have visibility and control over all network traffic in / out of all containers.

• **Container Platform Network Plugin**: Each container platform (e.g., Docker, Kubernetes) has its own plugin model for how external networking platforms integrate. In the case of Docker, each Linux node runs a process (cilium-docker) that handles each Docker libnetwork call and passes data / requests on to the main Cilium Agent.

In addition to the components that run on each Linux container host, Cilium leverages a key-value store to share data between Cilium Agents running on different nodes. The currently supported key-value stores are:

• etcd
• consul
• local storage (golang hashmap)

**Cilium Agent**

The Cilium agent (cilium-agent) runs on each Linux container host. At a high-level, the agent accepts configuration that describes service-level network security and visibility policies. It then listens to events in the container runtime to learn when containers are started or stopped, and it creates custom BPF programs which the Linux kernel uses to control all network access in / out of those containers. In more detail, the agent:

• Exposes APIs to allow operations / security teams to configure security policies (see below) that control all communication between containers in the cluster. These APIs also expose monitoring capabilities to gain additional visibility into network forwarding and filtering behavior.

• Gathers metadata about each new container that is created. In particular, it queries identity metadata like container / pod labels, which are used to identify endpoints in Cilium security policies.

• Interacts with the container platforms network plugin to perform IP address management (IPAM), which controls what IPv4 and IPv6 addresses are assigned to each container. The IPAM is managed by the agent in a shared pool between all plugins which means that the Docker and CNI network plugin can run side by side allocating a single address pool.

• Combines its knowledge about container identity and addresses with the already configured security and visibility policies to generate highly efficient BPF programs that are tailored to the network forwarding and security behavior appropriate for each container.

• Compiles the BPF programs to bytecode using clang/LLVM and passes them to the Linux kernel to run for all packets in / out of the container’s virtual ethernet device(s).

**Cilium CLI Client**

The Cilium CLI Client (cilium) is a command-line tool that is installed along with the Cilium Agent. It gives a command-line interface to interact with all aspects of the Cilium Agent API. This includes inspecting Cilium’s state about each network endpoint (i.e., container), configuring and viewing security policies, and configuring network monitoring behavior.
Linux Kernel BPF

Berkeley Packet Filter (BPF) is a Linux kernel bytecode interpreter originally introduced to filter network packets, e.g. tcpdump and socket filters. It has since been extended with additional data structures such as hashtable and arrays as well as additional actions to support packet mangling, forwarding, encapsulation, etc. An in-kernel verifier ensures that BPF programs are safe to run and a JIT compiler converts the bytecode to CPU architecture specific instructions for native execution efficiency. BPF programs can be run at various hooking points in the kernel such as for incoming packets, outgoing packets, system calls, kprobes, etc.

BPF continues to evolve and gain additional capabilities with each new Linux release. Cilium leverages BPF to perform core datapath filtering, mangling, monitoring and redirection, and requires BPF capabilities that are in any Linux kernel version 4.8.0 or newer. On the basis that 4.8.x is already declared end of life and 4.9.x has been nominated as a stable release we recommend to run at least kernel 4.9.17 (the latest current stable Linux kernel as of this writing is 4.10.x).

Cilium is capable of probing the Linux kernel for available features and will automatically make use of more recent features as they are detected.

Linux distros that focus on being a container runtime (e.g., CoreOS, Fedora Atomic) typically already ship kernels that are newer than 4.8, but even recent versions of general purpose operating systems such as Ubuntu 16.10 ship fairly recent kernels. Some Linux distributions still ship older kernels but many of them allow installing recent kernels from separate kernel package repositories.

For more detail on kernel versions, see: Prerequisites - Linux Kernel Version.

Key-Value Store

The Key-Value (KV) Store is used for the following state:

- Policy Identities: list of labels <=> policy identity identifier
- Global Services: global service id to VIP association (optional)
- Encapsulation VTEP mapping (optional)

To simplify things in a larger deployment, the key-value store can be the same one used by the container orchestrator (e.g., Kubernetes using etcd). In single node Cilium deployments used for basic testing / learning, Cilium can use a local store implemented as a golang hash map, avoiding the need to setup a dedicated KV store.

Address Management

Building microservices on top of container orchestrations platforms like Docker and Kubernetes means that application architects assume the existence of core platform capabilities like service discovery and service-based load-balancing to map between a logical service identifier and the IP address assigned to the containers / pods actually running that service. This, along with the fact that Cilium provides network security and visibility based on container identity, not addressing, means that Cilium can keep the underlying network addressing model extremely simple.

Cluster IP Prefixes and Container IP Assignment

With Cilium, all containers in the cluster are connected to a single logical Layer 3 network, which is associated a single cluster wide address prefix. This means that all containers or endpoint connected to Cilium share a single routable subnet. Hence, all endpoints have the capability of reaching each other with two routing operations performed (one routing operation is performed on both the origin and destination container host). Cilium supports IPv4 and IPv6 addressing in parallel, i.e. each container can be assigned an IPv4 and IPv6 address and these addresses can be used exchangeably.
The simplest approach is to use a private address space for the cluster wide address prefix. However there are scenarios where choosing a publicly routable addresses is preferred, in particular in combination with IPv6 where acquiring a large routeable addressing subnet is possible. (See the next section on IP Interconnectivity).

Each container host is assigned a node prefix out of the cluster prefix which is used to allocate IPs for local containers. Based on this, Cilium is capable of deriving the container host IP address of any container and automatically create a logical overlay network without further configuration. See section Overlay Routing for additional details.

### IPv6 IP Address Assignment

Cilium allocates addresses for local containers from the /48 IPv6 prefix called the cluster prefix. If left unspecified, this prefix will be f00d::/48. Within that prefix, a /96 prefix is dedicated to each container host in the cluster. Although the default prefix will enable communication within an isolated environment, the prefix is not publicly routable. It is strongly recommended to specify a public prefix owned by the user using the --node-addr option.

If no node address is specified, Cilium will try to generate a unique node prefix by using the first global scope IPv4 address as a 32 bit node identifier, e.g. f00d:0:0:0:<ipv4-address>::/96. Within that /96 prefix, each node will independently allocate addresses for local containers.

Note that only 16 bits out of the /96 node prefix are currently used when allocating container addresses. This allows to use the remaining 16 bits to store arbitrary connection state when sending packets between nodes. A typical use for the state is direct server return.

Based on the node prefix, two node addresses are automatically generated by replacing the last 32 bits of the address with 0:0 and 0:ffff respectively. The former is used as the next-hop address for the default route inside containers, i.e. all packets from a container will be sent to that address for further routing. The latter represents the Linux stack and is used to reach the local network stack, e.g. Kubernetes health checks.

TODO: I’d like to know what the logic to assign addresses. Especially, are those addresses assigned sequentially? Are they randomly chosen from available addresses in the prefix? What is the delay before an IPv6 address is reused? Is all that information persisted? Where? Is there really no risk of assigning the same IPv6 address twice?

**Example**

Cluster prefix: f00d::/48

Node A prefix: f00d:0:0:0:A:A::/96
Node A address: f00d:0:0:0:A:A:0:0/128
Container on A: f00d:0:0:0:A:A:0:1111/128

Node B prefix: f00d:0:0:0:B:B::/96
Node B address: f00d:0:0:0:B:B:0:0/128
Container on B: f00d:0:0:0:B:B:0:2222/128

### IPv4 IP Address Assignment

Cilium will allocate IPv4 addresses to containers out of a /16 node prefix. This prefix can be specified with the --ipv4-range option. If left unspecified, Cilium will try and generate a unique prefix using the format 10.X.0.0/16 where X is replaced with the last byte of the first global scope IPv4 address discovered on the node. This generated prefix is relatively weak in uniqueness so it is highly recommended to always specify the IPv4 range.

The address 10.X.0.1 is reserved and represents the local node.
IP Interconnectivity

When thinking about base IP connectivity with Cilium, it's useful to consider two different types of connectivity:

- Container-to-Container Connectivity
- Container Communication with External Hosts

Container-to-Container Connectivity

In the case of connectivity between two containers inside the same cluster, Cilium is in full control over both ends of the connection. It can thus transmit state and security context information between two container hosts by embedding the information in encapsulation headers or even unused bits of the IPv6 packet header. This allows Cilium to transmit the security context of where the packet originates from which allows tracing back which container labels are assigned to the origin container.

Note: As the packet headers contain security sensitive information, it is highly recommended to either encrypt all traffic or run Cilium in a trusted network environment.

There are two possible approaches to performing network forwarding for container-to-container traffic:

- **Overlay Routing**: In this mode, the network connecting the container hosts together does not need to be aware of the node prefix or the IP addresses of containers. Instead, a virtual overlay network is created on top of the existing network infrastructure by creating tunnels between containers hosts using encapsulation protocols such as VXLAN, GRE, or Geneve. This minimizes the requirements on the underlying network infrastructure. The only requirement in this mode is for containers hosts to be able to reach each other by UDP (VXLAN/Geneve) or IP/GRE. As this requirement is typically already met in most environments, this mode usually does not require additional configuration from the user. Cilium can deterministically map from any container IP address to the corresponding node IP address, Cilium can look at the destination IP address of any packet destined to a container, and then use encapsulation to send the packet directly to the IP address of the node running that container. The destination node then decapsulates the packet, and delivers it to the local container. Because overlay routing requires no configuration changes in the underlying network, it is often the easiest approach to adopt initially.

- **Direct Routing**: In this mode, Cilium will hand all packets which are not addresses to a local container and not addresses to the local node to the Linux stack causing it to route the packet as it would route any other non-local packet. As a result, the network connecting the Linux node hosts must be aware that each of the node IP prefixes are reachable by using the node’s primary IP address as an L3 next hop address. In the case of a traditional physical network this would typically involve announcing each node prefix as a route using a routing protocol within the datacenter. Cloud providers (e.g., AWS VPC, or GCE Routes) provide APIs to achieve the same result.

Regardless of the option chosen, the container itself has no awareness of the underlying network it runs on, it only contains a default route which points to the IP address of the container host. Given the removal of the routing cache in the Linux kernel, this reduces the amount of state to keep to the per connection flow cache (TCP metrics) which allows to terminate millions of connections in each container.

Container Communication with External Hosts

Container communication with the outside world has two primary modes:

- Containers exposing API services for consumption by hosts outside of the container cluster.
Containers making outgoing connections. Examples include connecting to 3rd-party API services like Twilio or Stripe as well as accessing private APIs that are hosted elsewhere in your enterprise datacenter or cloud deployment.

In the “Direct Routing” scenario described above, if container IP addresses are routable outside of the container cluster, communication with external hosts requires little more than enabling L3 forwarding on each of the Linux nodes.

External Connectivity with Overlay Routing

However, in the case of “Overlay Routing”, accessing external hosts requires additional configuration.

In the case of containers accepting inbound connections, such services are likely exposed via some kind of load-balancing layer, where the load-balancer has an external address that is not part of the Cilium network. This can be achieved by having a load-balancer container that both has a public IP on an externally reachable network and a private IP on a Cilium network. However, many container orchestration frameworks, like Kubernetes, have built in abstractions to handle this “ingress” load-balancing capability, which achieve the same effect that Cilium handles forwarding and security only for “internal” traffic between different services.

Containers that simply need to make outgoing connections to external hosts can be addressed by configuring each Linux node host to masquerade connections from containers to IP ranges other than the cluster prefix (IP masquerading is also known as Network Address Port Translation, or NAPT). This approach can be used even if there is a mismatch between the IP version used for the container prefix and the version used for Node IP addresses.

Security

Cilium provides security on multiple levels. Each can be used individually or combined together.

- **Identity based Connectivity Access Control:** Connectivity policies between endpoints (Layer 3), e.g. any endpoint with label \texttt{role=frontend} can connect to any endpoint with label \texttt{role=backend}.
- Restriction of accessible ports (Layer 4) for both incoming and outgoing connections, e.g. endpoint with label \texttt{role=frontend} can only make outgoing connections on port 443 (https) and endpoint \texttt{role=backend} can only accept connections on port 443 (https).
- Fine grained access control on application protocol level to secure HTTP and remote procedure call (RPC) protocols, e.g the endpoint with label \texttt{role=frontend} can only perform the REST API call \texttt{GET /userdata/[0-9]+}, all other API interactions with \texttt{role=backend} are restricted.

Currently on the roadmap, to be added soon:
- **Authentication:** Any endpoint which wants to initiate a connection to an endpoint with the label \texttt{role=backend} must have a particular security certificate to authenticate itself before being able to initiate any connections. See GH issue 502 for additional details.
- **Encryption:** Communication between any endpoint with the label \texttt{role=frontend} to any endpoint with the label \texttt{role=backend} is automatically encrypted with a key that is automatically rotated. See GH issue 504 to track progress on this feature.

Identity based Connectivity Access Control

Container management systems such as Kubernetes deploy a networking model which assigns an individual IP address to each pod (group of containers). This ensures simplicity in architecture, avoids unnecessary network address translation (NAT) and provides each individual container with a full range of port numbers to use. The logical consequence
of this model is that depending on the size of the cluster and total number of pods, the networking layer has to manage a large number of IP addresses.

Traditionally security enforcement architectures have been based on IP address filters. Let’s walk through a simple example: If all pods with the label `role=frontend` should be allowed to initiate connections to all pods with the label `role=backend` then each cluster node which runs at least one pod with the label `role=backend` must have a corresponding filter installed which allows all IP addresses of all `role=frontend` pods to initiate a connection to the IP addresses of all local `role=backend` pods. All other connection requests should be denied. This could look like this: If the destination address is `10.1.1.2` then allow the connection only if the source address is one of the following `[10.1.2.2,10.1.2.3,20.4.9.1]`.

Every time a new pod with the label `role=frontend` or `role=backend` is either started or stopped, the rules on every cluster node which run any such pods must be updated by either adding or removing the corresponding IP address from the list of allowed IP addresses. In large distributed applications, this could imply updating thousands of cluster nodes multiple times per second depending on the churn rate of deployed pods. Worse, the starting of new `role=frontend` pods must be delayed until all servers running `role=backend` pods have been updated with the new security rules as otherwise connection attempts from the new pod could be mistakenly dropped. This makes it difficult to scale efficiently.

In order to avoid these complications which can limit scalability and flexibility, Cilium entirely separates security from network addressing. Instead, security is based on the identity of a pod, which is derived through labels. This identity can be shared between pods. This means that when the first `role=frontend` pod is started, Cilium assigns an identity to that pod which is then allowed to initiate connections to the identity of the `role=backend` pod. The subsequent start of additional `role=frontend` pods only requires to resolve this identity via a key-value store, no action has to be performed on any of the cluster nodes hosting `role=backend` pods. The starting of a new pod must only be delayed until the identity of the pod has been resolved which is a much simpler operation than updating the security rules on all other cluster nodes.

![Diagram of Cilium's security model](image)

**What is an Endpoint Identity?**

The identity of an endpoint is derived based on the labels associated with the pod or container. When a pod or container is started, Cilium will create an endpoint based on the event received by the container runtime to represent the pod or container on the network. As a next step, Cilium will resolve the identity of the endpoint created. Whenever the labels of the pod or container change, the identity is reconfirmed and automatically modified as required.

Not all labels associated with a container or pod are meaningful when deriving the security identity. Labels may be
used to store metadata such as the timestamp when a container was launched. Cilium requires to know which labels are meaningful and are subject to being considered when deriving the identity. For this purpose, the user is required to specify a list of string prefixes of meaningful labels. The standard behavior is to include all labels which start with the prefix `id`, e.g. `id.service1`, `id.service2`, `id.groupA.service44`. The list of meaningful label prefixes can be specified when starting the cilium agent, see *Cilium Agent Command Line Options*.

**Special Identities**

All endpoints which are managed by Cilium will be assigned an identity. In order to allow communication to network endpoints which are not managed by Cilium, special identities exist to represent those. Special reserved identities are prefixed with the string `reserved:`.

<table>
<thead>
<tr>
<th>Identity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved:host</td>
<td>The host network namespace on which the pod or container is running.</td>
</tr>
<tr>
<td>reserved:world</td>
<td>Any network endpoint outside of the cluster</td>
</tr>
</tbody>
</table>


**Identity Management in the Cluster**

Identities are valid in the entire cluster which means that if several pods or containers are started on several cluster nodes, all of them will resolve and share a single identity if they share the identity relevant labels. This requires coordination between cluster nodes.

The operation to resolve an endpoint identity is performed with the help of the distributed key-value store which allows to perform atomic operations in the form *generate a new unique identifier if the following value has not been seen before*. This allows each cluster node to create the identity relevant subset of labels and then query the key-value store to derive the identity. Depending on whether the set of labels has been queried before, either a new identity will be created, or the identity of the initial query will be returned.
Policy Enforcement

All security policies are described assuming stateful policy enforcement for session based protocols. This means that the intent of the policy is to describe allowed direction of connection establishment. If the policy allows $A \rightarrow B$ then reply packets from $B$ to $A$ are automatically allowed as well. However, $B$ is not automatically allowed to initiate connections to $A$. If that outcome is desired, then both directions must be explicitly allowed.

Security policies are primarily enforced at ingress which means that each cluster node verifies all incoming packets and determines whether the packet is allowed to be transmitted to the intended endpoint. Policy enforcement also occurs at egress if required by the specific policy, e.g. a Layer 7 policy restricting outgoing API calls.

Layer 3 policies are currently not enforced at egress to avoid the complexity of resolving the destination endpoint identity before sending out the packet. Instead, the identity of the source endpoint is embedded into the packet.

In order to enforce identity based security in a multi host cluster, the identity of the transmitting endpoint is embedded into every network packet that is transmitted in between cluster nodes. The receiving cluster node can then extract the identity and verify whether a particular identity is allowed to communicate with any of the local endpoints.

Default Security Policy

If no policy is loaded, the default behaviour is to allow all communication unless policy enforcement has been explicitly enabled. As soon as the first policy rule is loaded, policy enforcement is enabled automatically and any communication must then be white listed or the relevant packets will be dropped.

Similar, if an endpoint is not subject to an $L4$ policy, communication from and to all ports is permitted. Associating at least one $L4$ policy to an endpoint will block all connectivity to ports unless explicitly allowed.

Orchestration System Specifics

Kubernetes

Cilium regards each deployed Pod as an endpoint with regards to networking and security policy enforcement. Labels associated with pods can be used to define the identity of the endpoint.

When two pods communicate via a service construct, then the labels of the origin pod apply to determine the identity.

Policy Language

The security policy can be specified in the following formats:

- The Kubernetes NetworkPolicy specification which offers to configure a subset of the full Cilium security. For fun see Kubernetes Network Policies for details on how to configure Kubernetes network policies. It is possible to define base rules using the Kubernetes specification and then extend these using additional Cilium specific rules.

- The Cilium policy language as described below. In addition to the what the Kubernetes NetworkPolicy spec supports, the Cilium language allows to implement Layer 7 filtering, deny rules, and hierarchical rules for delegation and precedence purposes. Cilium also provides egress enforcement for Layer 4 and Layer 7 rules.

The data format used by the Cilium policy language is JSON. Additional formats may be supported in the future.

Policy consists of a list of rules:

```json
{
    "rules": [{ rule1, rule2, rule3 }]
}
```
Policy Rules

Multiple types of policy rules are supported, all types following the simple template:

- **coverage**: A list of labels which the endpoint must carry.
- **rule**: A type specific rule, the following rule types have been implemented:
  - **Allow/Requires**: Connectivity policy, e.g. allow a pod to talk to another pod
  - **L4**: L4 connectivity policy

**Example:**

The following example describes a rule which applies to all endpoints which carry the label `id.service1`.

```
[{
   "coverage": ["role=backend"],
   "allow": allowData
}]
```

**Allow Rules**

This is the simplest rule type. The rule defines a list of labels which are allowed to consume whatever endpoints are covered by the coverage.

If an endpoint transmits to another endpoint and the communication is not permitted by at least one `allow` rule, all packets of the connection will be dropped.

**Note:** Packet drops can be introspected by running the `cilium monitor` tool which logs each dropped packet including metadata such as the reason (policy denied) and the source and destination identity.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coverage</td>
<td>Array of labels</td>
<td>List of labels which must match in order for this rule to be applied.</td>
</tr>
<tr>
<td>allow</td>
<td>Array of labels</td>
<td>List of labels which are allowed to initiate a connection to any endpoint covered by coverage.</td>
</tr>
</tbody>
</table>

allow:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>string</td>
<td>{ “accept”,“always-accept”,“deny” }</td>
</tr>
<tr>
<td>label</td>
<td>label</td>
<td>Allowed or denied label</td>
</tr>
</tbody>
</table>

A short form is available as alternative to the above verbose JSON syntax:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coverage</td>
<td>Array of strings</td>
<td>List of labels which must match in order for this rule to be applied.</td>
</tr>
<tr>
<td>allow</td>
<td>Array of strings</td>
<td>List of labels which are allowed to initiate a connection to any endpoint covered by coverage. The action is “accept” unless the label has the prefix / in which case the action is “deny”.</td>
</tr>
</tbody>
</table>

**Example:**

The following simple example using the form allows pods with the label `role=frontend` to consume pods with the label `role=backend`:
The following example using the short form allows all pods with the label `role=frontend` to consume pods with the label `role=backend` unless the frontend pod carries the label `user=joe`:

```
{
    "coverage": ["role=backend"],
    "allow": ["role=frontend", "!user=joe"]
}
```

The special `always-allow` action is useful in combination with hierarchical policy trees. It allows to define `allow` rules which cannot be overruled by child policy nodes. See Hierarchical Rules for additional information on policy tree and their precedence model.

The following example shows a child node `role` which contains a rule which disallows access from `role=frontend` to `role=backend`. However, the parent node `root` access by using `always-accept`.

```
{
    "name": "root",
    "rules": [{
        "coverage": ["role=backend"],
        "allow": [{
            "action": "always-accept",
            "label": { "key": "role=frontend" } }
        }],
    },
    "children": {
        "role": {
            "rules": [{
                "coverage": ["role=backend"],
                "allow": ["!role=frontend"]
            }]
        }
    }
}
```

### Requires Rules

`Requires` rules allow to define a list of additional labels which require to be present in the sending endpoint order for an allow rule to take effect. A `requires` rule itself does not grant permissions for consumption, it merely imposes additional constraints. At least one `allow` rule is always required.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coverage</td>
<td>Array of labels</td>
<td>List of labels which must match in order for this rule to be applied.</td>
</tr>
<tr>
<td>requires</td>
<td>Array of labels</td>
<td>List of labels which must be present in any transmitting endpoint desiring to connect to any endpoint covered by coverage.</td>
</tr>
</tbody>
</table>

If an endpoint transmits to another endpoint and the communication is not permitted because at least one of the required labels is not present, then the same behaviour applied as if it would lack an `allow` rule.

```
{
    "coverage": ["role=backend"],
    "allow": ["role=frontend", "!user=joe"]
}
```
The example above extends the existing allow rule with two additional requires rules. The first rule says that if an endpoint carries the label env=qa then the consuming endpoint also needs to carry the label env=qa. The second rule does the same for the label env=prod. The requires rules allows for simple segmentation of existing rules into multiple environments or groups.

Layer 4 Rules

The L4 rule allows to impose Layer 4 restrictions on endpoints. It can be applied to either incoming or outgoing connections. An L4 by itself does not allow communication, it must be combined with an allow rule to establish basic connectivity.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coverage</td>
<td>Array of labels</td>
<td>List of labels which must match in order for this rule to be applied.</td>
</tr>
<tr>
<td>in-ports</td>
<td>Array of l4-policy</td>
<td>Layer 4 policy for any incoming connection to an endpoint covered by coverage.</td>
</tr>
<tr>
<td>out-ports</td>
<td>Array of l4-policy</td>
<td>Layer 4 policy for any outgoing connection to an endpoint covered by coverage.</td>
</tr>
</tbody>
</table>

**l4-policy:**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>port</td>
<td>integer</td>
<td>Allowed destination port</td>
</tr>
<tr>
<td>protocol</td>
<td>string</td>
<td>Allowed protocol {&quot;tcp&quot;, &quot;udp&quot;} (optional)</td>
</tr>
<tr>
<td>l7-parser</td>
<td>string</td>
<td>Name of Layer 7 parser. If set, causes traffic to be inspected based on rules. (optional)</td>
</tr>
<tr>
<td>l7-rules</td>
<td>Array of string</td>
<td>Array of rules passed into Layer 7 parser (optional). See Layer 7 Rules</td>
</tr>
</tbody>
</table>

The following example shows how to restrict Layer 4 communication of any endpoint carrying the label role=frontend and restrict incoming connections to TCP on port 80 or port 443. Outgoing connections must also be TCP and are restricted to port 8080.

```json
[{
    "coverage": ["role=frontend"],
    "14": [{
        "in-ports": [
            { "port": 80, "protocol": "tcp" },
            { "port": 443, "protocol": "tcp" }
        ],
        "out-ports": [{
            "port": 8080, "protocol": "tcp"
        }]
    }
}]
```
Layer 7 Rules

Layer 7 rules are currently limited to IPv4. Policies can be applied for both incoming and outgoing requests. The enforcement point is defined by the location of the rules in either the “in-ports” or “out-ports” field of the Layer 4 policy rule.

Unlike Layer 3 and Layer 4 policies, violation of Layer 7 rules does not result in packet drops. Instead, if possible, an access denied message such as an **HTTP 403 access denied** is sent back to the sending endpoint.

TODO: describe rules

Hierarchical Rules

In order to allow implementing precedence and priority of rules. Policy rules can be organized in the form of a tree. This tree consists of policy nodes based on the following definition:

- **Name** [string (optional)] Relative name of the policy node. If omitted, then “root” is assumed and rules belong to the root node. Must be unique across all siblings which are attached to the same parent.

- **Rules** [array of rules] List of rules, see **Policy Rules**

**Children**: Map with node entries (optional) Map holding children policy nodes. The name of each child policy node is prefixed with the name of its parent policy node using a . delimiter, e.g. a node child attached to the root node will have the absolute name root.child.

```json
{
    "name": "root",
    "rules": [{ rule1, rule2, rule3 }]
    "children": {
        "child1": {
            "rules": [{ rule1, rule2, rule3 }]
        },
        "child2": {
            "rules": [{ rule1, rule2, rule3 }]
        }
    }
}
```

Automatic coverage of child nodes

A key property of child policy nodes is that their name implies an implicit coverage. The absolute name of the policy node with the root. prefix omitted acts as an implicit coverage which is applied to all rules of the node.

**Example**: A node k8s which is attached to the node io will have the absolute name root.io.k8s. Rules of the node will only apply if the endpoint in question carries a label which starts with the prefix io.k8s.

Additionally, any rules of a child node may only cover labels which share the prefix of the absolute node path. This means that a child id.foo cannot contain a rule which covers the label id.bar.example but it can contain a rule which covers the label id.foo.example.

Unlike an arbitrary label selector attached to each node, this property ensures that a parent node always covers all endpoints of all its children which is essential to keep precedence rules simple as described in the next section.
Precedence Rules

1. Within a single policy node, a deny rule always overwrites any conflicting allow rules. If a label is both denied and allowed, it will always be denied.

2. If a node allows a label and a child node later denies the label then the label will be denied unless the allow rule is a *always-allow* rule in which case the parent always takes precedence.

Merging of Nodes

TODO

Policy Repository

Policy rules imported into the Cilium agent are not shared with other compute nodes and are only enforced within the boundaries of the compute node. In order to enforce security policies across an entire cluster, one of the following options can be applied to distribute security policies across all cluster nodes:

- Use of Kubernetes NetworkPolicy objects to define the policy. NetworkPolicy objects are automatically distributed to all worker nodes and the Cilium agent will import them automatically. (TODO: Describe option to use third-party objects to distribute native Cilium policy).

- Use of a configuration management system such as chef, puppet, ansible, cfengine to automatically import a policy into all agents. (TODO: link to guide as soon as one exists.)

- Use of a git tree to maintain the policy in combination with a post-merge hook which automatically imports the policy. (TODO: Write & link to guide)

- Use of a distributed filesystem shared across all cluster node in combination with a filesystem watcher which invokes *cilium import* upon detection of any change.

Integration with Container Platforms

Cilium is deeply integrated with container platforms like Docker or Kubernetes. This enables Cilium to perform network forwarding and security using a model that maps direction to notions of identity (e.g., labels) and service abstractions that are native to the container platform.

In this section, we will provide more detail on how Cilium integrates with Docker and Kubernetes.

Docker supports network plugins via the libnetwork plugin interface.

When using Cilium with Docker, one creates a single logical Docker network of type “cilium” and with an IPAM-driver of type “cilium”, which delegates control over IP address management and network connectivity to Cilium for all containers attached to this network for both IPv4 and IPv6 connectivity. Each Docker container gets an IP address from the node prefix of the node running the container.

When deployed with Docker, each Linux node runs a “cilium-docker” agent, which receives libnetwork calls from Docker and then communicates with the Cilium Agent to control container networking.

Security policies controlling connectivity between the Docker containers can be written in terms of the Docker container labels passed to Docker when creating the container. These policies can be created/updated via communication directly with the Cilium agent, either via API or using the Cilium CLI client.

When deployed with Kubernetes, Cilium provides four core Kubernetes networking capabilities:

- Direct pod-to-pod network inter-connectivity.
• Service-based load-balancing for pod-to-pod inter-connectivity (i.e., a kube-proxy replacement).
• Identity-based security policies for all (direct and service-based) Pod-to-Pod inter-connectivity.
• External-to-Pod service-based load-balancing (referred to as “Ingress” in Kubernetes)

The Kubernetes documentation contains more background on the Kubernetes Networking Model and Kubernetes Network Plugins.

In Kubernetes, containers are deployed within units referred to as Pods, which include one or more containers reachable via a single IP address. With Cilium, each Pod gets an IP address from the node prefix of the Linux node running the Pod. In the absence of any network security policies, all Pods can reach each other.

Pod IP addresses are typically local to the Kubernetes cluster. If pods need to reach services outside the cluster as a client, the Kubernetes nodes are typically configured to IP masquerade all traffic sent from containers to external prefix.

Kubernetes has developed the Services abstraction which provides the user the ability to load balance network traffic to different pods. This abstraction allows the pods reaching out to other pods by a single IP address, a virtual IP address, without knowing all the pods that are running that particular service.

Without Cilium, kube-proxy is installed on every node, watches for endpoints and services addition and removal on the kube-master which allows it to apply the necessary enforcement on iptables. Thus, the received and sent traffic from and to the pods are properly routed to the node and port serving for that service. For more information you can check out the Kubernetes user guide for Services.

Cilium loadbalancer acts on the same principles as kube-proxy, it watches for services addition or removal, but instead of doing the enforcement on the iptables, it updates bpf maps entries on each node. For more information, see the Pull Request.

TODO: describe benefits of BPF based load-balancer compared to kube-proxy iptables

Todo: Verify this

Kubernetes supports an abstraction known as Ingress that allows a Pod-based Kubernetes service to expose itself for access outside of the cluster in a load-balanced way. In a typical setup, the external traffic would be sent to a publicly reachable IP + port on the host running the Kubernetes master, and then be load-balanced to the pods implementing the current service within the cluster.

Cilium supports Ingress with TCP-based load-balancing. Moreover, it supports “direct server return”, meaning that reply traffic from the pod to the external client is sent directly, without needing to pass through the Kubernetes master host.

Todo: insert graphic showing LB + DSR.
This document describes how to install, configure, and troubleshoot Cilium in different deployment modes. It assumes you have already read and understood the components and concepts described in the Architecture Guide. This document focuses on a full deployment of Cilium within a datacenter or public cloud. If you are just looking for a simple way to experiment on your laptop, we highly recommend using our Vagrant environment:

**Vagrant**

**Using the provided Vagrantfile**

*Note:* You need to run Vagrant version 1.8.3 or later or you will run into issues booting the Ubuntu 16.10 base image.

To bring up a vagrant VM with Cilium plus dependencies installed, run:

```
$ contrib/vagrant/start.sh
```

This will create and run a vagrant VM based on the base box `cilium/ubuntu-16.10`. The box is currently available for the following providers:

- libvirt
- virtualbox

**Options**

The following environment variables can be set to customize the VMs brought up by vagrant:

- `NWORKERS=n`: Number of child nodes you want to start with the master, default 0.
- `RELOAD=1`: Issue a `vagrant reload` instead of `vagrant up`
• NFS=1: Use NFS for vagrant shared directories instead of rsync
• K8S=1: Build & install kubernetes on the nodes
• IPV4=1: Run Cilium with IPv4 enabled
• VAGRANT_DEFAULT_PROVIDER={virtualbox | libvirt | ...}

If you want to start the VM with cilium enabled with IPv4, with kubernetes installed and plus a worker, run:

```bash
$ IPV4=1 K8S=1 NWORKERS=1 contrib/vagrant/start.sh
```

If you have any issue with the provided vagrant box cilium/ubuntu-16.10 or need a different box format, you may build the box yourself using the packer scripts

**Manual installation**

Alternatively you can import the vagrant box cilium/ubuntu-16.10 directly and manually install Cilium:

```bash
$ vagrant init cilium/ubuntu-16.10
$ vagrant up
$ vagrant ssh [...]
$ cd go/src/github.com/cilium/cilium/
$ make
$ sudo make install
$ sudo cp contrib/upstart/* /etc/init/
$ sudo usermod -a -G cilium vagrant
$ sudo service cilium restart`
```

**Prerequisites - Linux Kernel Version**

Since Cilium builds on the foundation of BPF, the kernel version running on the Linux node that runs containers and Cilium must support a modern version of BPF. For practical purposes, this means that the kernel must be 4.8.0 or newer.

For a version of Linux designed to be used as a container runtime, this is often the case already, as such distributions frequently update their default kernels. For example:

- **CoreOS stable** is already at 4.9.9 as of March 2017.
- **Fedora atomic** is at 4.9.12 as of March 2017.

General purpose Linux distros are less likely to come with a 4.8+ kernel by default. As of March 2017, **Ubuntu 16.10** is the most widely used general purpose distro that runs a 4.8+ kernel by default. However, many other popular distros have the ability to optionally install a kernel with version 4.8 or newer.

**Installing Cilium**

Cilium consists of an agent plus additional optional integration plugins which must be installed on all servers which will run containers.

The easiest way to leverage the Cilium agent on your Linux container node is to install it as a container itself. This section will cover that option for both vanilla Docker deployments as well as Kubernetes. It will also describe how to build and install from source in the case that you need to run Cilium directly on the Linux container host without a container.
Installing Cilium using Docker Compose

Below is an example of using Docker Compose to deploy the Cilium agent and the Cilium Docker libnetwork plugin.

Note: for multi-host deployments using a key-value store, you would want to update this template to point cilium to a central key-value store.

```yaml
version: '2'
services:
  cilium:
    container_name: cilium
    image: cilium/cilium:cilium-ubuntu-16-04
    command: cilium-agent --debug -d $INTERFACE -c 127.0.0.1:8500
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - /var/run/cilium:/var/run/cilium
      - /run/docker/plugins:/run/docker/plugins
      - /sys/fs/bpf:/sys/fs/bpf
    network_mode: "host"
    cap_add:
      - "NET_ADMIN"
    privileged: true
  cilium_docker:
    container_name: cilium-docker-plugin
    image: cilium/cilium:cilium-ubuntu-16-04
    command: cilium-docker -D
    volumes:
      - /var/run/cilium:/var/run/cilium
      - /run/docker/plugins:/run/docker/plugins
    network_mode: "host"
    cap_add:
      - "NET_ADMIN"
    privileged: true
    depends_on:
      - cilium
```

Installing Cilium using Kubernetes Daemon Sets

If you are using Kubernetes, you can automatically have a pod running cilium distributed to each Linux container node in the cluster using Kubernetes Daemon Sets.

Here is an example Daemon Set definition:

```yaml
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: cilium-net-controller
spec:
  template:
    metadata:
      labels:
        name: cilium-net-controller
        io.cilium.admin.daemon-set: "cilium"
    spec:
      nodeSelector:
        with-network-plugin: cilium
```
container:
  - image: cilium/cilium:cilium-ubuntu-16-04
    imagePullPolicy: Always
    name: cilium-net-daemon
    command: [ "/home/with-cni.sh", "--debug", "daemon", "run" ]
    args:
      - "-t"
      - "vxlan"
      - "--etcd-config-path"
      - "/var/lib/cilium/etcd-config.yml"
      - "--k8s-kubeconfig-path"
      - "/var/lib/kubelet/kubeconfig"
    env:
      - name: "K8S_NODE_NAME"
        valueFrom:
          fieldRef:
            fieldPath: spec.nodeName
    volumeMounts:
      - name: cilium-run
        mountPath: /var/run/cilium
      - name: cni-path
        mountPath: /tmp/cni/bin
      - name: bpf-maps
        mountPath: /sys/fs/bpf
      - name: docker-socket
        mountPath: /var/run/docker.sock
        readOnly: true
      - name: etcd-config
        mountPath: /var/lib/cilium/etcd-config.yml
        readOnly: true
      - name: kubeconfig-path
        mountPath: /var/lib/kubelet/kubeconfig
        readOnly: true
      - name: kubeconfig-cert
        mountPath: /var/lib/kubernetes/ca.pem
        readOnly: true
    securityContext:
      capabilities:
        add:
        - "NET_ADMIN"
      privileged: true
      hostNetwork: true
    volumes:
      - name: cilium-run
        hostPath:
          path: /var/run/cilium
      - name: cni-path
        hostPath:
          path: /opt/cni/bin
      - name: bpf-maps
        hostPath:
          path: /sys/fs/bpf
      - name: docker-socket
        hostPath:
          path: /var/run/docker.sock
      - name: etcd-config
        hostPath:
          path: /var/lib/cilium/etcd-config.yml

4.3. Installing Cilium
To deploy this pod to each cluster node, first give each node in the cluster a label like "with-network-plugin=cilium"

```
kubectl label node worker0 with-network-plugin=cilium
kubectl label node worker1 with-network-plugin=cilium
kubectl label node worker2 with-network-plugin=cilium
```

Save the above daemon set definition to a file named cilium-ds.yaml, and then create this daemon set with kubectl:

```
kubectl create -f cilium-ds.yaml
```

Kubernetes will deploy a copy of the daemon set to each node with the correct "with-network-plugin=cilium" label. You can watch the progress of this deployment using:

```
kubectl get daemonset cilium-net-controller
```

### Build + Install From Source

Installing Cilium from a container is recommended. If you need to build / install Cilium directly on the container Linux node, there are additional required dependencies beyond a 4.8.0+ Linux kernel:

- clang+LLVM >=3.7.1. Please note that in order to use clang 3.9.x, the kernel version requirement is >= 4.9.17
- iproute2 >= 4.8.0: https://www.kernel.org/pub/linux/utils/net/iproute2/
- (recommended) Linux kernel >= 4.9.17. Use of a 4.9.17 kernel or later will ensure compatibility with clang > 3.9.x

Download the Cilium source code, and run `make install`. This will install cilium binaries in your `bindir` and all required additional runtime files in `libdir/cilium`.

Templates for integration into service management systems such as systemd and upstart can be found in the `contrib` directory.

For example:

```
make install
sudo cp contrib/upstart/* /etc/init/
service cilium start
```

### Container Node Network Configuration

The networking configuration required on your Linux container node depends on the IP interconnectivity model in use and whether the deployment requires containers in the cluster to reach or be reached by resources outside the cluster. For more details, see the Architecture Guide’s section on IP Interconnectivity.
**Overlay Mode - Container-to-Container Access**

With overlay mode, container-to-container access does not require additional network configuration on the Linux container node, as overlay connectivity is handled by Cilium itself, and the physical network only sees IP traffic destined to/from the Linux node IP address.

The use of Overlay Mode is configured by passing a `--tunnel` or `-t` flag to the Cilium indicating the type of encapsulation to be used. Valid options include `vxlan` and `geneve`.

**Direct Mode - Container-to-Container Access**

In direct mode, container traffic is sent to the underlying network unencapsulated, and thus that network must understand how to route a packet to the right destination Linux node running the container.

Direct mode is used if no `-t` or `--tunneling` flag is passed to the Cilium agent at startup.

Cilium automatically enables IP forwarding in Linux when direct mode is configured, but it is up to the container cluster administrator to ensure that each routing element in the underlying network has a route that describe each node IP as the IP next hop for the corresponding node prefix.

If the underlying network is a physical datacenter network, this can be achieved by running a routing daemon on each Linux node that participates in the datacenter’s routing protocol, such as bird, zebra or radvd. Configuring this setup is beyond the scope of this document.

If the underlying network is a virtual network in a public cloud, that cloud provider likely provides APIs to configure the routing behavior of that virtual network (e.g., AWS VPC Route Tables or GCE Routes). These APIs can be used to associate each node prefix with the appropriate next hop IP each time a container node is added to the cluster.

An example using GCE Routes for this is available [here](#).

**External Network Access**

By default with Cilium, containers use IP addresses that are private to the cluster. This is very common in overlay mode, but may also be the case even if direct mode is being used. In either scenario, if a container with a private IP should be allowed to make outgoing network connections to resources elsewhere in the data center or on the public Internet, the Linux node should be configured to perform IP masquerading, also known as network address port translation (NAPT), for all traffic destined from a container to the outside world.

An example of configuring IP masquerading for IPv6 is:

```
ip6tables -t nat -I POSTROUTING -s f00d::/112 -o em1 -j MASQUERADE
```

This will masquerade all packets with a source IP in the cluster prefix `beef::/64` with the public IPv6 address of the Linux nodes primary network interface `em1`. If you change your cluster IP address or use IPv4 instead of IPv6, be sure to update this command accordingly.

**Testing External Connectivity**

IPv6 external connectivity can be tested with:

```
ip -6 route get `host -t aaaa www.google.com | awk '{print $5}'`
ping6 www.google.com
```

If the default route is missing, your VM may not be receiving router advertisements. In this case, the default route can be added manually:
The following tests connectivity from a container to the outside world:

```
$ sudo docker run --rm -ti --net cilium -l client cilium/demo-client ping6 www.google.com
PING www.google.com(zrh04s07-in-x04.1e100.net) 56 data bytes
64 bytes from zrh04s07-in-x04.1e100.net: icmp_seq=1 ttl=56 time=7.84 ms
64 bytes from zrh04s07-in-x04.1e100.net: icmp_seq=2 ttl=56 time=8.63 ms
64 bytes from zrh04s07-in-x04.1e100.net: icmp_seq=3 ttl=56 time=8.83 ms
```

Note that an appropriate policy must be loaded or policy enforcement will drop the relevant packets. An example policy can be found in `examples/policy/test/` which will allow the above container with the label `io.cilium` to be reached from world scope. To load and test:

```
$ cilium policy import examples/policy/test/test.policy
$ cilium policy allowed -s reserved:world -d io.cilium
```

## Configuring Cilium to use a Key-Value Store

Cilium can use both Consul and etcd as a key-value store. See `Cilium Agent Command Line Options` for the command-line options to configure both options.

## Container Platform Integrations

### Docker

#### Configuring Cilium as a Docker Network Plugin

As described above, the Cilium installation process creates a `cilium-docker` which implements the plugin logic. When launched, the cilium-docker binary automatically registers itself with the local Docker daemon.

The cilium-docker binary also communicates with the main Cilium Agent via the agent’s UNIX domain socket (`/var/run/cilium/cilium.sock`), so the plugin binary must have permissions to send / receive calls to this socket.

#### Network Creation

As isolation and segmentation is enforced based on Docker container labels, all containers can be attached to a single Docker network (this is the recommended configuration). Please note that IPv6 must be enabled on the network as the IPv6 address is also the unique identifier for each container:

```
$ docker network create --ipv6 --subnet ::1/112 --driver cilium --ipam-driver cilium
$ docker run --net cilium hello-world
```

#### Running a Container

Any container attached to a Cilium managed network will automatically have networking managed by Cilium. For example:
$ docker run --net cilium hello-world

## Kubernetes

### API Server Configuration

With Kubernetes there is only one implicit logical network for pods, so rather than creating a network, you start the Kubernetes API server with a prefix matching your Cilium prefix (e.g., 
`--service-cluster-ip-range="f00d:1::/112"`)  

**Important note**: The `service-cluster-ip-range` is currently limited to a single address family. This means that unless you are running Cilium with `--disable-ipv4`, the `service-cluster-ip-range` must be set to an IPv4 range. This should get resolved once Kubernetes starts supporting multiple IP addresses for a single pod.

TODO: do we need to recommend installing security policies that enable kube-dns, etc?

### Container Node / Kubelet Configuration

#### Enabling the Cilium and Loopback CNI Plugins

Create cni configuration file to tell the kubelet that it should use the cilium cni plugin:

```
sudo mkdir -p /etc/cni/net.d
sudo sh -c "echo "\{
  "name": "cilium",
  "type": "cilium-cni",
  "mtu": 1450
}"
" > /etc/cni/net.d/10-cilium-cni.conf"
```

Since Kubernetes v1.3.5 the user needs to install the loopback cni plugin:

```
sudo mkdir -p /opt/cni
wget https://storage.googleapis.com/kubernetes-release/network-plugins/cni-\n  07a8a28637e97b22eb8dfe710eaea1344f69d16e.tar.gz
sudo tar -xvf cni-07a8a28637e97b22eb8dfe710eaea1344f69d16e.tar.gz -C /opt/cni
```

Make two changes to the kubelet systemd unit file:

- include an [ExecPre] block to mount the BPF filesystem: `ExecPre=/bin/mount bpffs /sys/fs/bpf -t bpffs`
- include a flag instructing the kubelet to use CNI plugins: `--network-plugin=cni`

An example systemd file with these changes is below:

```
sudo sh -c 'cat > /etc/systemd/system/kubelet.service <<"EOF"
[Unit]
Description=Kubernetes Kubelet
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=docker.service
Requires=docker.service

[Service]
ExecPre=/bin/mount bpffs /sys/fs/bpf -t bpffs
EOF"'
```

### 4.6. Container Platform Integrations
Disabling Kube-proxy

Additionally, you should disable the local kube-proxy running on each container Node, as Cilium performs this function itself.

TODO: include command for disabling kube-proxy

Cilium Agent Command Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>config</td>
<td>config file</td>
<td>$HOME/ciliumd.yaml</td>
</tr>
<tr>
<td>consul</td>
<td>Consul agent address</td>
<td></td>
</tr>
<tr>
<td>debug</td>
<td>Enable debug messages</td>
<td>false</td>
</tr>
<tr>
<td>device</td>
<td>Ethernet device to snoop on</td>
<td></td>
</tr>
<tr>
<td>disable-connttrack</td>
<td>Disable connection tracking</td>
<td>false</td>
</tr>
<tr>
<td>enable-policy</td>
<td>Enable policy enforcement</td>
<td>false</td>
</tr>
<tr>
<td>docker</td>
<td>Docker socket endpoint</td>
<td></td>
</tr>
<tr>
<td>etcd</td>
<td>etcd agent address</td>
<td></td>
</tr>
<tr>
<td>etcd-config-path</td>
<td>absolute path to the etcd config</td>
<td></td>
</tr>
<tr>
<td>enable-tracing</td>
<td>enable policy tracing</td>
<td></td>
</tr>
<tr>
<td>nat46-range</td>
<td>IPv6 range to map IPv4 addresses to</td>
<td></td>
</tr>
<tr>
<td>k8s-api-server</td>
<td>Kubernetes api address server</td>
<td></td>
</tr>
<tr>
<td>k8s-kubeconfig-path</td>
<td>Absolute path to the kubeconfig file</td>
<td></td>
</tr>
<tr>
<td>k8s-prefix</td>
<td>Key-value store prefix used by k8s</td>
<td></td>
</tr>
<tr>
<td>keep-config</td>
<td>When restoring state, keeps containers’ configuration in place</td>
<td>false</td>
</tr>
<tr>
<td>label-prefix-file</td>
<td>file with label prefixes cilium Cilium should use for policy</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
Table 4.1 – continued from previous page

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>labels</td>
<td>list of label prefixes Cilium should use for policy</td>
<td></td>
</tr>
<tr>
<td>logstash</td>
<td>enable logstash integration</td>
<td>false</td>
</tr>
<tr>
<td>logstash-agent</td>
<td>logstash agent address and port</td>
<td>127.0.0.1:8080</td>
</tr>
<tr>
<td>node-address</td>
<td>IPv6 address of the node</td>
<td></td>
</tr>
<tr>
<td>restore</td>
<td>Restore state from previously running version of the agent</td>
<td>false</td>
</tr>
<tr>
<td>keep-templates</td>
<td>do not restore templates from binary</td>
<td>false</td>
</tr>
<tr>
<td>state-dir</td>
<td>path to store runtime state</td>
<td></td>
</tr>
<tr>
<td>lib-dir</td>
<td>path to store runtime build env</td>
<td></td>
</tr>
<tr>
<td>socket-path</td>
<td>path for agent unix socket</td>
<td></td>
</tr>
<tr>
<td>lb</td>
<td>enables load-balancing mode on interface ‘device’</td>
<td></td>
</tr>
<tr>
<td>disable-ipv4</td>
<td>disable IPv4 mode</td>
<td>false</td>
</tr>
<tr>
<td>ipv4-range</td>
<td>IPv4 prefix</td>
<td></td>
</tr>
<tr>
<td>tunnel</td>
<td>Overlay/tunnel mode (vxlan/geneve)</td>
<td>vxlan</td>
</tr>
<tr>
<td>bpf-root</td>
<td>Path to mounted BPF filesystem</td>
<td></td>
</tr>
<tr>
<td>access-log</td>
<td>Path to HTTP access log</td>
<td></td>
</tr>
</tbody>
</table>

Cilium CLI Commands

TODO: cover Cilium CLI commands

Troubleshooting

TODO: troubleshooting

- describe locations of log files
- describe tools used for debugging
CHAPTER 5

BPF and XDP Reference Guide

Note: This documentation section is targeted at developers and users who want to understand BPF and XDP in great technical depth. While reading this reference guide may help broaden your understanding of Cilium, it is not a requirement to use Cilium. Please refer to the Getting Started Guide and Architecture Guide for a higher level introduction.

BPF is a highly flexible and efficient “virtual machine”-like construct in the Linux kernel allowing to execute bytecode at various hook points in a safe manner. It is used in a number of Linux kernel subsystems, most prominently networking, tracing and security (e.g. sandboxing).

While BPF has existed since 1992, this document covers the extended Berkley Paket Filter (eBPF) version which has first appeared in Kernel 3.18 and obsoletes the original version which is being referred to as “classic” BPF (cBPF) these days. cBPF is known to many as being the packet filter language use by tcpdump. Nowadays, the Linux kernel runs eBPF only and loaded cBPF bytecode is transparently translated into an eBPF representation in the kernel before program execution. This documentation will generally refer to the term BPF unless explicit differences between eBPF and cBPF are being pointed out.

Even though the name Berkley Packet Filter hints at a packet filtering specific purpose, the instruction set is generic and flexible enough these days that there are many use cases for BPF apart from networking. See Projects using BPF for a list of projects which use BPF.

Cilium uses BPF heavily in its data path, see Architecture Guide for further information. The goal of this chapter is to provide an BPF reference guide in order to gain understanding of BPF its networking specific use including loading BPF programs with tc (traffic control) and XDP (eXpress Data Path), and to aide developing Cilium’s BPF templates.

BPF Architecture

BPF does not define itself by only providing its instruction set, but also by offering further infrastructure around it such as maps that act as efficient key / value stores, helper functions to interact with and leverage kernel functionality, tail calls for calling into other BPF programs, security hardening primitives, a pseudo file system for pinning objects (maps, programs), and infrastructure for allowing BPF to be offloaded, for example, to a network card.
LLVM provides an BPF back end, such that tools like clang can be used to compile C into an BPF object file, which can then be loaded into the kernel. BPF is deeply tied into the Linux kernel and allows for full programmability without sacrificing native kernel performance.

Last but not least, also the kernel subsystems making use of BPF are part of BPF’s infrastructure. The two main subsystems discussed throughout this document are tc and XDP where BPF programs can be attached to. XDP BPF programs are attached at the earliest networking driver stage and trigger a run of the BPF program upon packet reception. By definition, this achieves the best possible packet processing performance since packets cannot get processed at an even earlier point in software. Driver support is necessary in order to use XDP BPF programs, though. However, tc BPF programs don’t need any driver support and can be attached to receive and transmit paths of any networking device, including virtual ones such as veth devices since they hook later in the kernel stack compared to XDP. Apart from tc and XDP programs, there are various other kernel subsystems as well that use BPF such as tracing (kprobes, uprobes, tracepoints, etc).

The following subsections provide further details on individual aspects of the BPF architecture.

**Instruction Set**

BPF is a general purpose RISC instruction set and was originally designed with the goal to write programs in a subset of C that can be compiled into BPF instructions through a compiler back end (e.g., LLVM), such that the kernel can later on map them through an in-kernel JIT compiler into native opcodes for optimal execution performance inside the kernel.

The advantages for pushing these instructions into the kernel are:

- Making the kernel programmable without having to cross kernel / user space boundaries. For example, BPF programs related to networking as in the case of Cilium, can implement flexible container policies, load balancing and other means without having to move packets to user space and back into the kernel. State between BPF programs and kernel / user space can still be shared through maps whenever needed.

- Given the flexibility of a programmable data path, programs can be heavily optimized for performance also by compiling out features that are not required for the use cases the program solves. F.e., if a container does not require IPv4, then the BPF program can be built to only deal with IPv6 in order to save resources in the fast-path.

- In case of networking (e.g., tc and XDP), BPF programs can be updated atomically without having to restart the kernel, system services or containers, and without traffic interruptions. Furthermore, any program state can also be maintained throughout updates via BPF maps.

- BPF provides a stable ABI towards user space, and does not require any third party kernel modules, for example. BPF is a core part of the Linux kernel that is shipped everywhere, and guarantees that existing BPF programs keep running with newer kernel versions. This guarantee is the same guarantee that the kernel provides for system calls with regard to user space applications.

- BPF programs work in concert with the kernel, they make use of existing kernel infrastructure (e.g., drivers, netdevices, tunnels, protocol stack, sockets) and tooling (e.g., iproute2) as well as the safety guarantees that the kernel provides. Unlike kernel modules, BPF programs are verified through an in-kernel verifier in order to ensure that they cannot crash the kernel, always terminate, etc. XDP programs, for example, reuse the existing in-kernel drivers and operate on the provided DMA buffers containing the packet frames without exposing them or an entire driver to user space as in other models. Moreover, XDP programs reuse the existing stack instead of bypassing it. BPF can be considered as generic “glue code” to kernel facilities for crafting programs to solve specific use cases.

The execution of an BPF program inside the kernel is always event driven! For example, a networking device that has an BPF program attached on its ingress path will trigger the execution of the program once a packet is received, a kernel address that has a kprobes with an BPF program attached will trap once the code at that address gets executed, invoke the kprobes callback function for instrumentation and testing which then triggers the execution of the BPF program attached to it.
BPF consists of eleven 64 bit registers with 32 bit subregisters, a program counter and a 512 byte large BPF stack space. Registers are named \texttt{r0 - r10}. The operating mode is 64 bit by default, the 32 bit subregisters can only be accessed through special ALU operations. The 32-bit lower subregisters zero-extend into 64-bit when they are being written to.

Register \texttt{r10} is the only register which is read-only and contains the frame pointer address in order to access the BPF stack space. The remaining \texttt{r0 - r9} registers are general purpose and of read/write nature.

An BPF program can call into a predefined helper function, which is defined by the core kernel (never by modules). The BPF calling convention is defined as follows:

- \texttt{r0} contains the return value of a helper function call.
- \texttt{r1 - r5} hold arguments from the BPF program to the kernel helper function.
- \texttt{r6 - r9} are callee saved registers that will be preserved on helper function call.

The BPF calling convention is generic enough that it maps directly to x86, arm64 and other ABIs, thus all BPF registers map one to one to HW CPU registers, so that a JIT only needs to issue a call instruction, but no additional extra moves for placing function arguments. This calling convention was modeled to cover common call situations without having a performance penalty. Calls with 6 or more arguments are currently not supported. The helper functions in the kernel that are dedicated to BPF (\texttt{BPF_CALL_0()} to \texttt{BPF_CALL_5()} functions) are specifically designed with this convention in mind.

Register \texttt{r0} is also the register that contains the exit value for the BPF program. The semantics of the exit value are defined by the type of program. Furthermore, when handing execution back to the kernel, the exit value is passed as a 32 bit value.

Registers \texttt{r1 - r5} are scratch registers, meaning the BPF program needs to either spill them to the BPF stack or move them to callee saved registers if these arguments are to be reused across multiple helper function calls. Spilling means that the variable in the register is moved to the BPF stack. The reverse operation of moving the variable from the BPF stack to the register is called filling. The reason for spilling/filling is due to limited number of registers.

Upon entering execution of an BPF program, register \texttt{r1} initially contains the context for the program. The context is the input argument for the program (similar to \texttt{argc/argv} pair for a typical C program). BPF is restricted to work on a single context. The context is defined by the program type, for example, a networking program can have a kernel representation of the network packet (\texttt{skb}) as the input argument.

The general operation of BPF is 64 bit to follow the natural model of 64-bit architectures in order to perform pointer arithmetics, pass pointers but also pass 64 bit values into helper functions, and to allow for 64 bit atomic operations.

The maximum instruction limit per program is restricted to 4096 BPF instructions, which, by design, means that any program will terminate quickly. Although the instruction set contains forward as well as backward jumps, the in-kernel BPF verifier will forbid loops such that termination is always guaranteed. Since BPF programs run inside the kernel, the verifier’s job is to make sure that these are safe to run, not affecting the system’s stability. This means that from an instruction set point of view, loops can be implemented, but the verifier will restrict that. However, there is also a concept of tail calls that allows for one BPF program to jump into another one. This, too, comes with an upper nesting limit of 32 calls, and is usually used to decouple parts of the program logic, for example, into stages.

The instruction format is modeled as two operand instructions, which helps mapping BPF instructions to native instructions during JIT phase. The instruction set is of fixed size, meaning every instruction has 64 bit encoding. Currently, 87 instructions have been implemented and the encoding also allows to extend the set with further instructions when needed. The instruction encoding of a single 64 bit instruction is defined as a bit sequence (from MSB to LSB) of \texttt{op:8, dst_reg:4, src_reg:4, off:16, imm:32}. \texttt{off} and \texttt{imm} is of signed type. The encodings are part of the kernel headers and defined in \texttt{linux/bpf.h} header, which also includes \texttt{linux/bpf_common.h}.

\texttt{op} defines the actual operation to be performed. Most of the encoding for \texttt{op} has been reused from cBPF. The operation can be based on register or immediate operands. The encoding of \texttt{op} itself provides information on which mode to use (\texttt{BPF_X} for denoting register-based operations, and \texttt{BPF_K} for immediate-based operations respectively). In case of the latter, the destination operand is always a register. Both \texttt{dst_reg} and \texttt{src_reg} provide additional
information about the register operands to be used (e.g., r0 - r9) for the operation. off is used in some instructions to provide a relative offset, for example, for addressing the stack or other buffers available to BPF (e.g., map values, packet data, etc.), or jump targets in jump instructions. imm contains a constant / immediate value.

The available op instructions can be categorized into various instruction classes. These classes are also encoded inside the op field. The op field is divided into (from MSB to LSB) code:4, source:1 and class:3. class is the more generic instruction class, code specifies a specific operational code inside that class, and source tells whether the source operand is a register or an immediate value. Possible instruction classes are:

- **BPF_LD, BPF_LDX:** Both classes are for load operations. BPF_LD is used for loading a double word as a special instruction spanning two instructions due to the imm:32 split, and for byte / half-word / word loads of packet data. The latter was carried over from cBPF mainly in order to keep cBPF to BPF translations efficient, since they have optimized JIT code. For native BPF these packet load instructions are less relevant nowadays. BPF_LDX class holds instructions for byte / half-word / word / double-word loads out of memory. Memory in this context is generic and could be stack memory, map value data, packet data, etc.

- **BPF_ST, BPF_STX:** Both classes are for store operations. Similar to BPF_LDX the BPF_STX is the store counterpart and is used to store the data from a register into memory, which, again, can be stack memory, map value, packet data, etc. BPF_STX also holds special instructions for performing word and double-word based atomic add operations, which can be used for counters, for example. The BPF_ST class is similar to BPF_STX by providing instructions for storing data into memory only that the source operand is an immediate value.

- **BPF_ALU, BPF_ALU64:** Both classes contain ALU operations. Generally, BPF_ALU operations are in 32 bit mode and BPF_ALU64 in 64 bit mode. Both ALU classes have basic operations with source operand that is register-based and an immediate-based counterpart. Supported by both are add (+), sub (-), and (\&), or (\|), left shift (\ll), right shift (\rr), xor (^), mul (*), div (/), mod (\%) operations. Also mov (\langle X \rangle := \langle Y \rangle) was added as a special ALU operation for both classes in both operand modes. BPF_ALU64 also contains a signed right shift. BPF_ALU additionally contains endianness conversion instructions for half-word / word / double-word on a given source register.

- **BPF_JMP:** This class is dedicated to jump operations. Jumps can be unconditional and conditional. Unconditional jumps simply move the program counter forward, thus that the next instruction to be executed relative to the current instruction is off + 1, where off is the constant offset encoded in the instruction. Since off is signed, the jump can also be performed backwards as long as it does not create a loop and is within program bounds. Conditional jumps operate on both, register-based and immediate-based source operands. If the condition in the jump operations results in true, then a relative jump to off + 1 is performed, otherwise when false the next instruction (0 + 1) is performed. This fall-through jump logic differs compared to cBPF and allows for better branch prediction as it fits the CPU branch predictor logic more naturally. Available conditions are jeq (==), jne (!=), jgt (\rangle), jge (\geq), jsgt (signed \rangle), jsge (signed \geq), jset (jump if DST & SRC). Apart from that, there are three special jump operations within this class: the exit instruction which will leave the BPF program and return the current value in r0 as a return code, the call instruction, which will issue a function call into one of the available BPF helper functions, and a hidden tail call instruction, which will jump into a different BPF program.

The Linux kernel ships with an BPF interpreter that executes programs assembled in BPF instructions. Even cBPF programs are translated into BPF programs transparently in the kernel with the exception that an architecture still ships with a cBPF JIT and has not yet migrated to an BPF JIT.

Currently x86_64, arm64, ppc64, s390x and sparc64 architectures come with an in-kernel eBPF JIT compiler.

All BPF handling such as loading of programs into the kernel or creation of BPF maps is managed through a central bpf() system call. It is also used for managing map entries (lookup / update / delete), and making programs as well as maps persistent in the BPF file system through pinning.
Helper Functions

Helper functions are a concept that lets BPF programs consult a core kernel defined set of function calls in order to retrieve/push data from/to the kernel. Available helper functions may differ for each BPF program type, for example, BPF programs attached to sockets are only allowed to call into a subset of helpers as opposed to BPF programs attached to the tc layer. Encapsulation and decapsulation helpers for lightweight tunneling constitute an example of functions that are only available to lower tc layers, event output helpers for pushing notifications to user space for tc and XDP programs.

Each helper function is implemented with a commonly shared function signature similar to system calls. The signature is defined as:

```
u64 fn(u64 r1, u64 r2, u64 r3, u64 r4, u64 r5)
```

The calling convention as described in the previous section applies for all BPF helper functions.

The kernel abstracts helper function into macros BPF_CALL_0() to BPF_CALL_5() that are similar to those of system calls. The following example is an extract from a helper function which updates map elements by calling into the corresponding map implementation callbacks:

```
BPF_CALL_4(bpf_map_update_elem, struct bpf_map *, map, void *, key,
           void *, value, u64, flags)
{
    WARN_ON_ONCE(!rcu_read_lock_held());
    return map->ops->map_update_elem(map, key, value, flags);
}

const struct bpf_func_proto bpf_map_update_elem_proto = {
    .func = bpf_map_update_elem,
    .gpl_only = false,
    .ret_type = RET_INTEGER,
    .arg1_type = ARG_CONST_MAP_PTR,
    .arg2_type = ARG_PTR_TO_MAP_KEY,
    .arg3_type = ARG_PTR_TO_MAP_VALUE,
    .arg4_type = ARG_ANYTHING,
};
```

There are various advantages with this approach: while cBPF overloaded its load instructions in order to fetch data at an impossible packet offset to invoke auxiliary helper functions, each cBPF JIT needed to implement support for such a cBPF extension. In case of eBPF, each newly added helper function will be JIT compiled in a transparent and efficient way, meaning that the JIT compiler only needs to emit a call instruction since the register mapping is made in such a way that BPF register assignments already match the underlying architecture’s calling convention. This allows for easily extending the core kernel with new helper functionality.

Mentioned function signature also allows the verifier to perform type checks. The above struct bpf_func_proto is used to hand all the necessary information that is needed to know about the helper to the verifier, so the verifier can make sure that expected types from the helper match with the current contents of the BPF program’s analyzed registers.

Argument types can range from passing in any kind of value up to restricted contents such as a pointer/size pair for the BPF’s stack buffer, which the helper should read from or write to. In the latter case, the verifier can also perform additional checks, for example, whether the buffer was initialized previously.

Maps

Maps are efficient key/value stores that reside in kernel space. They can be accessed from a BPF program in order to keep state among multiple BPF program invocations. They can also be accessed through file descriptors from user
space and can be arbitrarily shared with other BPF programs or user space applications.

BPF programs that share maps with each other are not required to be of the same program type, for example, tracing programs can share maps with networking programs. A single BPF program can currently access up to 64 different maps directly.

Map implementations are provided by the core kernel. There are generic maps with per-CPU and non-per-CPU flavour that can read / write arbitrary data, but there are also a few non-generic maps that are used along with helper functions.

Generic maps that are currently available:

- BPF_MAP_TYPE_HASH
- BPF_MAP_TYPE_ARRAY
- BPF_MAP_TYPE_PERCPU_HASH
- BPF_MAP_TYPE_PERCPU_ARRAY
- BPF_MAP_TYPE_LRU_HASH
- BPF_MAP_TYPE_LRU_PERCPU_HASH
- BPF_MAP_TYPE_LPM_TRIE

Non-generic maps currently in the kernel:

- BPF_MAP_TYPE_PROG_ARRAY
- BPF_MAP_TYPE_PERF_EVENT_ARRAY
- BPF_MAP_TYPE_CGROUP_ARRAY
- BPF_MAP_TYPE_STACK_TRACE
- BPF_MAP_TYPE_ARRAY_OF_MAPS
- BPF_MAP_TYPE_HASH_OF_MAPS

TODO: further coverage of maps and their purpose

Object Pinning

BPF maps and programs act as a kernel resource and can only be accessed through file descriptors, backed by anonymous inodes in the kernel. Advantages, but also a number of disadvantages come along with them:

User space applications can make use of most file descriptor related APIs, file descriptor passing for Unix domain sockets work transparently, etc, but at the same time, file descriptors are limited to a processes’ lifetime, which makes possibilities like map sharing rather cumbersome to realize.

Thus, it brings a number of complications for certain use cases such as iproute2, where tc or XDP sets up and loads the program into the kernel and terminates itself eventually. With that, also access to maps are unavailable from user space side, where it would otherwise have been useful, for example, when maps are shared between ingress and egress locations of the data path. Also, third party applications may wish to monitor or update map contents during BPF program runtime.

To overcome this limitation, a minimal kernel space BPF file system has been implemented, where BPF map and programs can be pinned to, a process called object pinning. The BPF system call has therefore been extended with two new commands that can pin (BPF_OBJ_PIN) or retrieve (BPF_OBJ_GET) a previously pinned object.

For instance, tools such as tc make use of this infrastructure for sharing maps on ingress and egress. The BPF related file system is not a singleton, it does support multiple mount instances, hard and soft links, etc.
Tail Calls

Another concept that can be used with BPF is called tail calls. Tail calls can be seen as a mechanism that allows one BPF program to call another, without returning back to the old program. Such a call has minimal overhead as unlike function calls, it is implemented as a long jump, reusing the same stack frame.

Such programs are verified independently of each other, thus for transferring state, either per-CPU maps as scratch buffers or in case of tc programs, skb fields such as the cb[] area must be used.

Only programs of the same type can be tail called, and they also need to match in terms of JIT compilation, thus either JIT compiled or only interpreted programs can be invoked, but not mixed together.

There are two components involved for realizing tail calls: the first part needs to setup a specialized map called program array (BPF_MAP_TYPE_PROG_ARRAY) that can be populated by user space with key / values where values are the file descriptors of the tail called BPF programs, the second part is a bpf_tail_call() helper where the context, a reference to the program array and the lookup key is passed to. The kernel then inlines this helper call directly into a specialized BPF instruction. Such a program array is currently write-only from user space side.

The kernel looks up the related BPF program from the passed file descriptor and atomically replaces program pointers at the given map slot. When no map entry has been found at the provided key, the kernel will just “fall through” and continue execution of the old program with the instructions following after the bpf_tail_call(). Tail calls are a powerful utility, for example, parsing network headers could be structured through tail calls. During runtime, functionality can be added or replaced atomically, and thus altering the BPF program’s execution behaviour.

JIT

The 64 bit x86_64, arm64, ppc64, s390x and sparc64 architectures all ship with an in-kernel eBPF JIT compiler (mips64 is work in progress at this time), also all of them are feature equivalent and can be enabled through:

```
# echo 1 > /proc/sys/net/core/bpf_jit_enable
```

The 32 bit arm, mips, ppc and sparc architectures currently have a cBPF JIT compiler. The mentioned architectures still having a cBPF JIT as well as all remaining architectures supported by the Linux kernel which do not have a BPF JIT compiler at all need to run eBPF programs through the in-kernel interpreter.

In the kernel’s source tree, eBPF JIT support can be easily determined through issuing a grep for HAVE_EBPFF_JIT:

```
# git grep HAVE_EBPFF_JIT arch/
arch/arm64/Kconfig: select HAVE_EBPFF_JIT
arch/powerpc/Kconfig: select HAVE_EBPFF_JIT if PPC64
arch/s390/Kconfig: select HAVE_EBPFF_JIT if PACK_STACK && HAVE_MARCH_Z196
arch/sparc/Kconfig: select HAVE_EBPFF_JIT if SPARC64
arch/x86/Kconfig: select HAVE_EBPFF_JIT if X86_64
```

Hardening

BPF locks the entire BPF interpreter image (struct bpf_prog) as well as the JIT compiled image (struct bpf_binary_header) in the kernel as read-only during the program’s life-time in order to prevent the code from potential corruptions. Any corruption happening at that point, for example, due to some kernel bugs will result in a general protection fault and thus crash the kernel instead of allowing the corruption silently to happen.

Architectures that support setting the image memory as read-only can be determined through:
The option `CONFIG_ARCH_HAS_SET_MEMORY` is not configurable, such that this protection is always built-in. Other architectures might follow in the future.

In case of `/proc/sys/net/core/bpf_jit_harden` set to 1 additional hardening steps for the JIT compilation take effect for unprivileged users. This effectively trades off performance for them by decreasing a (potential) attack surface in case of untrusted users operating on the system. The decrease in program execution still results in better performance compared to switching to interpreter entirely.

Currently, enabling hardening will blind all user provided 32 bit and 64 bit constants from the BPF program when it gets JIT compiled in order to prevent JIT spraying attacks that inject native opcodes as immediate values. This is problematic as these immediate values reside in executable kernel memory, such that a jump that could be triggered from some kernel bug would jump to the start of the immediate value and then execute these as native instructions.

JIT constant blinding prevents that by randomizing the actual instruction, meaning the operation is transformed from a immediate based source operand to a register based one through rewriting the instruction by splitting the actual load of the value into two steps: 1) load of a blinded immediate value `rnd ^ imm` into a register, 2) xoring that register with `rnd` such that the original `imm` immediate then resides in the register and can be used for the actual operation. The example was provided for a load operation, but really all generic operations are blinded.

Example of JITing a program with hardening disabled:

```
# echo 0 > /proc/sys/net/core/bpf_jit_harden

fffffffffa034f5e9 + <x>:
[...]
39:  mov  $0xa8909090,%eax
3e:  mov  $0xa8909090,%eax
43:  mov  $0xa8ff3148,%eax
48:  mov  $0xa89081b4,%eax
4d:  mov  $0xa8900bb0,%eax
52:  mov  $0xa810e0c1,%eax
57:  mov  $0xa8908eb4,%eax
5c:  mov  $0xa89020b0,%eax
[...]
```

The same program gets constant blinded when loaded through BPF as an unprivileged user in the case hardening is enabled:

```
# echo 1 > /proc/sys/net/core/bpf_jit_harden

fffffffffa034f1e5 + <x>:
[...]
39:  mov  $0xe1192563,%r10d
3f:  xor  $0x4989b5f3,%r10d
46:  mov  %r10d,%eax
49:  mov  $0xb8296d93,%r10d
4f:  xor  $0x10b9fd03,%r10d
56:  mov  %r10d,%eax
59:  mov  $0x8c381146,%r10d
5f:  xor  $0x24c7200e,%r10d
66:  mov  %r10d,%eax
69:  mov  $0xeb2a830e,%r10d
```

5.1. BPF Architecture
The programs are both semantically the same, only that none of the original immediate values are visible anymore in
the disassembly.

At the same time, hardening also disabled any JIT kallsyms exposure for privileged users, so that kernel addresses are
not exposed to /proc/kallsyms.

**Offloads**

Networking programs in BPF, in particular for tc and XDP do have an offload-interface to hardware in the kernel in
order to execute BPF code directly on the NIC.

Currently, the nfp driver from Netronome has support for offloading BPF through a JIT compiler which translates
BPF instructions to an instruction set implemented against the NIC.

**Toolchain**

Current user space tooling, introspection facilities and kernel control knobs around BPF are discussed in this section.
Note, the tooling and infrastructure around BPF is still heavily evolving and thus may not provide a complete picture
of all available tools.

**LLVM**

LLVM is currently the only compiler suite that provides an BPF back end. gcc does not support BPF at this point.

The BPF back end was merged into LLVM’s 3.7 release. Major distributions enable the BPF back end by default when
they package LLVM, such that installing clang and llvm is sufficient on most recent distributions to start compiling C
into BPF object files.

The typical workflow is that BPF programs are written in C, compiled by LLVM into object / ELF files, that are parsed
by user space BPF ELF loaders (such as iproute2 or others), and pushed into the kernel through the BPF system call.
The kernel verifies the BPF instructions and JITs them, returning a new file descriptor for the program, which can then
be attached to a subsystem (e.g., networking). If supported, the subsystem could then further offload the BPF program
to hardware (e.g., NIC).

For LLVM, BPF target support can be checked, for example, through the following:

```
$ llc --version
LLVM (http://llvm.org/):
LLVM version 3.8.1
Optimized build.
Default target: x86_64-unknown-linux-gnu
Host CPU: skylake

Registered Targets:
```
By default, the `bpf` target uses the endianness of the CPU it compiles on, meaning, if the CPU’s endianness is little endian, the program is represented in little endian format as well, and if the CPU’s endianness is big endian, the program is represented in big endian. This also matches the runtime behavior of BPF, which is generic and uses the CPU’s endianness it runs on in order to not disadvantage architectures in any of the format.

For cross-compilation, the two targets `bpfeb` and `bpfel` were introduced, such that BPF programs can be compiled on a node running in one endianness (f.e., little endian on x86) and run on a node in another endianness format (f.e., big endian on arm). Note that the front end (clang) needs to run in the target endianness as well.

Using `bpf` as a target is the preferred way in situations where no mixture of endianness applies. For example, compilation on x86 results in the same output for the targets `bpf` and `bpfel` due to being little endian, therefore scripts triggering a compilation also do not have to be endian aware.

A minimal, stand-alone XDP drop program might look like the following (`xdp.c`):

```c
#include <linux/bpf.h>

#ifndef __section
#define __section(NAME) \
__attribute__((section(NAME), used))
#endif

__section("prog")
int xdp_drop(struct xdp_md *ctx)
{
    return XDP_DROP;
}

char __license[] __section("license") = "GPL";
```

It can then be compiled and loaded into the kernel as follows:

```
$ clang -O2 -Wall -target bpf -c xdp.c -o xdp.o
# ip link set dev em1 xdp obj xdp.o
```

For the generated object file LLVM (>= 3.9) uses the official BPF machine value, that is, `EM_BPF` (decimal: 247 | hex: `0xf7`). In this example, the program has been compiled with `bpf` target under x86, therefore LSB (as opposed to MSB) is shown regarding endianness:

```
$ file xdp.o
xdp.o: ELF 64-bit LSB relocatable, *unknown arch 0xf7* version 1 (SYSV), not stripped
```

`readelf -a xdp.o` will dump further information about the ELF file, which can sometimes be useful for introspecting generated section headers, relocation entries and the symbol table.

In the unlikely case where clang and LLVM needs to be compiled from scratch, the following commands can be used:

```
$ git clone http://llvm.org/git/llvm.git
$ cd llvm/tools
$ git clone --depth 1 http://llvm.org/git/clang.git
$ cd ..; mkdir build; cd build
$ cmake .. -DLLVM_TARGETS_TO_BUILD="BPF;X86" -DBUILD_SHARED_LIBS=OFF -DCMAKE_BUILD_TYPE=Release -DLLVM_BUILD_RUNTIME=OFF
```

5.2. Toolchain
$ make -j $(getconf _NPROCESSORS_ONLN)

$ ./bin/llc --version
LLVM (http://llvm.org/):
LLVM version x.y.zsvn
Optimized build.
Default target: x86_64-unknown-linux-gnu
Host CPU: skylake

Registered Targets:
  bpf - BPF (host endian)
bpef - BPF (big endian)
bpe1 - BPF (little endian)
x86 - 32-bit X86: Pentium-Pro and above
x86-64 - 64-bit X86: EM64T and AMD64

$ export PATH=$PWD/bin:$PATH  # add to ~/.bashrc

Make sure that --version mentions Optimized build., otherwise the compilation time for programs when having LLVM in debugging mode will significantly increase (f.e., by 10x or more).

For debugging, clang can generate the assembler output as follows:

$ clang -O2 -S -Wall -target bpf -c xdp.c -o xdp.S
$ cat xdp.S

.text
  .section prog,"ax",@progbits
  .globl xdp_drop
  .p2align 3

xdp_drop:  # @xdp_drop
  # BB#0:
    r0 = 1
  exit

  .section license,"aw",@progbits
  .globl __license  # @__license
__license:
  .asciz "GPL"

Furthermore, more recent LLVM versions (>= 4.0) can also store debugging information in dwarf format into the object file. This can be done through the usual workflow by adding -g for compilation.

$ clang -O2 -g -Wall -target bpf -c xdp.c -o xdp.o
$ llvm-objdump -S -no-show-raw-instr xdp.o

xdp.o: file format ELF64-BPF

Disassembly of section prog:
xdp_drop:
  | 0: r0 = 1
  | return XDP_DROP;
  1: exit

The llvm-objdump tool can then annotate the assembler output with the original C code that was used in the compilation. The trivial example in this case does not contain much C code, however, the line numbers shown as 0: and 1: correspond directly to the kernel’s verifier log.
This means that in case BPF programs get rejected by the verifier, `llvm-objdump` can help to correlate the instructions back to the original C code, which is highly useful for analysis.

```
# ip link set dev em1 xdp obj xdp.o verb

Prog section 'prog' loaded (5)!
- Type: 6
- Instructions: 2 (0 over limit)
- License: GPL

Verifier analysis:
0: (b7) r0 = 1
1: (95) exit
processed 2 insns
```

As can be seen in the verifier analysis, the `llvm-objdump` output dumps the same BPF assembler code as the kernel. Leaving out the `-no-show-raw-instruction` option will also dump the raw struct `bpf_insn` as hex in front of the assembly:

```
$ llvm-objdump -S xdp.o

xdp.o: file format ELF64-BPF

Disassembly of section prog:
xdp_drop:
  { 
  0: b7 00 00 00 01 00 00 00 r0 = 1
  ; return foo();
  l: 95 00 00 00 00 00 00 00 exit
```

For LLVM IR debugging, the compilation process for BPF can be split into two steps, generating a binary LLVM IR intermediate file `xdp.bc`, which can later on be passed to `llc`:

```
$ clang -O2 -Wall -emit-llvm -c xdp.c -o xdp.bc
$ llc xdp.bc -march=bpf -filetype=obj -o xdp.o
```

The generated LLVM IR can also be dumped in human readable format through:

```
$ clang -O2 -Wall -emit-llvm -S -c xdp.c -o -
```

Note that LLVM’s BPF back end currently does not support generating code that makes use of BPF’s 32 bit subregisters. Inline assembly for BPF is currently unsupported, too.

Furthermore, compilation from BPF assembly (e.g., `llvm-mc xdp.S -arch bpf -filetype=obj -o xdp.o`) is currently also not supported due to missing BPF assembly parser.

When writing C programs for BPF, there are a couple of pitfalls to be aware of compared to usual application development with C. The following items describe some of the differences for the BPF model:

1. **Everything needs to be inlined, there are no function or shared library calls available.**

   Shared libraries, etc., cannot be used with BPF. However, common library code that is used in BPF programs can be placed into header files and included into the main programs. For example, Cilium makes heavy use of this (see `bpf/lib`). However, this still allows for including header files, for example, from the kernel or other libraries and reuse their static inline functions or macros / definitions.

   Eventually LLVM needs to compile the entire code into a flat sequence of BPF instructions for a given program section. Best practice is to use an annotation like `__inline` for every library function as shown below. The
use of always_inline is recommended, since the compiler could still decide to uninline large functions that are only annotated as inline.

In case the latter happens, LLVM will generate a relocation entry into the ELF file, which BPF ELF loaders such as iproute2 cannot resolve and will thus throw an error since only BPF maps are valid relocation entries that loaders can process.

```c
#include <linux/bpf.h>

#ifndef __section
#define __section(NAME) \
   __attribute__((section(NAME), used))
#endif

#ifndef __inline
#define __inline \
   inline __attribute__((always_inline))
#endif

static __inline int foo(void) {
   return XDP_DROP;
}

__section("prog")
int xdp_drop(struct xdp_md *ctx) {
   return foo();
}

char __license[] __section("license") = "GPL";
```

2. **Multiple programs can reside inside a single C file in different sections.**

C programs for BPF make heavy use of section annotations. A C file is typically structured into 3 or more sections. BPF ELF loaders use these names to extract and prepare the relevant information in order to load the programs and maps through the bpf system call. For example, iproute2 uses maps and license as default section name to find meta data needed for map creation and the license for the BPF program, respectively. The latter is pushed into the kernel as well on program creation time, and enables some of the helper functions that are exposed as GPL only in case the program also holds a GPL compatible license, for example bpf_ktime_get_ns(), bpf_probe_read() and others.

The remaining section names are specific for BPF program code, for example, the below code has been modified to contain two program sections, ingress and egress. The toy example code demonstrates that both can share a map and common static inline helpers such as the account_data() function.

The xdp.c example has been modified to a tc.c example that can be loaded with tc and attached to a netdevice’s ingress and egress hook. It accounts the transferred bytes into a map called acc_map, which has two map slots, one for traffic accounted on the ingress hook, one on the egress hook.

```c
#include <linux/bpf.h>
#include <linux/pkt_cls.h>
#include <stdint.h>
#include <iproute2/bpf_elf.h>

#ifndef __section
#define __section(NAME) \
   __attribute__((section(NAME), used))
#endif
```
The example also demonstrates a couple of other things that are useful to be aware of when developing programs. The code includes kernel headers, standard C headers and an iproute2 specific header that contains the definition of `struct bpf_elf_map`. iproute2 has a common BPF ELF loader and as such the definition of `struct bpf_elf_map` is the very same for XDP and tc typed programs.

A `struct bpf_elf_map` entry defines a map in the program and contains all relevant information (such as key / value size, etc) that is needed in order to generate a map that is used from the two BPF
programs. The structure must be placed into the maps section, so that the loader can find it. There can be multiple such map declarations with different variable names, but all must be annotated with \_\_section("maps")

The \texttt{struct bpf\_elf\_map} is specific to iproute2. Different BPF ELF loaders can have different formats, for example, the libbpf in the kernel source tree which is mainly used by \texttt{perf} has a different specification. iproute2 guarantees backwards compatibility for \texttt{struct bpf\_elf\_map}. Cilium follows the iproute2 model.

The example also demonstrates how BPF helper functions are mapped into the C code and being used. Here, \texttt{map\_lookup\_elem()} is defined by mapping this function into the \texttt{BPF\_FUNC\_map\_lookup\_elem} enum value that is exposed as a helper in \texttt{linux/bpf.h}. When the program is later loaded into the kernel, the verifier checks whether the passed arguments are of the expected type and re-points the helper call into a real function call. Moreover, \texttt{map\_lookup\_elem()} also demonstrates how maps can be passed to BPF helper functions. Here, \texttt{\&acc\_map} from the maps section is passed as the first argument to \texttt{map\_lookup\_elem()}.

Since the defined array map is global, the accounting needs to use an atomic operation, which is defined as \texttt{lock\_xadd()}. LLVM maps \texttt{\_sync\_fetch\_and\_add()} as a built-in function to the BPF atomic add instruction, that is, \texttt{BPF\_STX | BPF\_XADD | BPF\_W} for word sizes.

Last but not least, the \texttt{struct bpf\_elf\_map} tells that the map is to be pinned as PIN\_GLOBAL\_NS. This means that tc will pin the map into the BPF pseudo file system as a node. By default, it will be pinned to \texttt{/sys/fs/bpf/tc/globals/acc\_map} for the given example. Due to the PIN\_GLOBAL\_NS, the map will be placed under \texttt{/sys/fs/bpf/tc/globals/}. \texttt{globals} acts as a global namespace that spans across object files. If the example would have used PIN\_OBJECT\_NS, then tc will create a directory that is local to the object file. For example, different C files with BPF code could have the same acc\_map definition as above with a PIN\_GLOBAL\_NS pinning. In that case, the map will be shared among BPF programs originating from various object files. PIN\_NONE would mean that the map is not placed into the BPF file system as a node, and would as a result not be accessible from user space after tc has quit. It would also mean that tc creates two separate map instances for each program, since it cannot retrieve a previously pinned map under that name. The acc\_map part from the mentioned path is the name of the map as specified in the source code.

Thus, upon below loading of the ingress program, tc will find that no such map exists in the BPF file system and creates a new one. Upon success, it will also pin the map, so that when the egress program is loaded through tc, it will find that such map already exists in the BPF file system and will reuse that for the egress program. The loader also makes sure in case maps exist with the same name that also their properties (key / value size, etc) match.

Just like tc can retrieve the same map, also third party applications can use the BPF\_OBJ\_GET command from the bpf system call in order to create a new file descriptor that points to the same map instance, which can then be used to lookup / update / delete map elements.

The code can be compiled and loaded via iproute2 as follows:

```bash
$ clang -O2 -Wall -target bpf -c tc.c -o tc.o
# tc qdisc add dev em1 clsact
# tc filter add dev eml ingress bpf da obj tc.o sec ingress
# tc filter add dev eml egress bpf da obj tc.o sec egress

# tc filter show dev eml ingress
filter protocol all pref 49152 bpf
filter protocol all pref 49152 bpf handle 0x1 tc.o:[ingress] direct-action_
   −tag c5f7825e5dac396f

# tc filter show dev eml egress
```

Chapter 5. BPF and XDP Reference Guide
As soon as packets pass the `em1` device, counters from the BPF map will be increased.

### 3. There are no global variables allowed.

For the same reasons as mentioned in point 1., BPF cannot have global variables as often used in normal C programs.

However, there is a work-around in that the program can simply use an BPF map of type `BPF_MAP_TYPE_PERCPU_ARRAY` with just a single slot of arbitrary value size. This works, because during execution, BPF programs are guaranteed to never get preempted by the kernel and therefore can use the single map entry as a scratch buffer for temporary data, for example, to extend beyond the stack limitation. This also works across tail calls, since it has the same guarantees with regards to preemption.

Otherwise, for holding state across multiple BPF program runs, normal BPF maps can be used.

### 4. There are no const strings or arrays allowed.

Defining `const` strings or other arrays in the BPF C program does not work for the same reasons as pointed out in 1. and 3., which is, that relocation entries will be generated in the ELF file that loaders will reject due to not being part of the ABI towards loaders (loaders also cannot fix up such entries as it would require large rewrites of the already compiled BPF sequence).

In future, LLVM might detect these occurrences and throw an error early to the user.

Helper functions such as `trace_printk()` can be worked around as follows:

```c
static void BPF_FUNC(trace_printk, const char *fmt, int fmt_size, ...);

#ifndef printk
#define printk(fmt, ...)  
  ({  
    char ____fmt[] = fmt;  
    trace_printk(____fmt, sizeof(____fmt), ##__VA_ARGS__);  
  })
#endif
```

The program can then use the macro naturally like `printk("skb len:%u\n", skb->len);`. The output will then be written to the trace pipe. `tc exec bpf dbg` can be used to retrieve the messages from there.

The use of the `trace_printk()` helper function has a couple of disadvantages and is thus not recommended for production usage. Constant strings like the "skb len:%u\n" need to be loaded into the

5.2. Toolchain
BPF stack each time the helper function is called, but also BPF helper functions are limited to a maximum of 5 arguments. This leaves room for only 3 additional variables that can be passed for dumping.

Therefore, while helpful for quick debugging, it is recommended (for networking programs) to use the \texttt{skb\_event\_output()} or the \texttt{xdp\_event\_output()} helper, respectively. They allow for passing custom structs from the BPF program to the perf event ring buffer along with an optional packet sample. For example, Cilium’s monitor makes use of these helpers in order to implement a debugging framework, notifications for network policy violations, etc. These helpers pass the data through a lockless memory mapped per-CPU \texttt{perf} ring buffer, and is thus significantly faster than \texttt{trace\_printk()}.

5. Use of LLVM built-in functions for \texttt{memset()}\texttt{/memcpy()}\texttt{/memmove()}\texttt{/memcmp()}.

Since BPF programs cannot perform any function calls other than those to BPF helpers, common library code needs to be implemented as inline functions. In addition, also LLVM provides some built-ins that the programs can use for constant sizes (here: \texttt{n}) which will then always get inlined:

\begin{verbatim}
#define memset(dest, chr, n) __builtin_memset((dest), (chr), (n))
#undef memset
#define memcpy(dest, src, n) __builtin_memcpy((dest), (src), (n))
#undef memcpy
#define memmove(dest, src, n) __builtin_memmove((dest), (src), (n))
#undef memmove
\end{verbatim}

The \texttt{memcmp()} built-in had some corner cases where inlining took not place due to an LLVM issue in the back end, and is therefore not recommended to be used until the issue is fixed.

6. There are no loops available.

The BPF verifier in the kernel checks that an BPF program does not contain loops by performing a depth first search of all possible program paths besides other control flow graph validations. The purpose is to make sure that the program is always guaranteed to terminate.

A very limited form of looping is available for constant upper loop bounds by using \texttt{#pragma unroll} directive. Example code that is compiled to BPF:

\begin{verbatim}
#pragma unroll
for (i = 0; i < IPV6_MAX_HEADERS; i++) {
  switch (nh) {
  case NEXTHDR_NONE:
    return DROP_INVALID_EXTHDR;
  case NEXTHDR_FRAGMENTS:
    return DROP_FRAG_NOSUPPORT;
  case NEXTHDR_HOP:
  case NEXTHDR_ROUTING:
  case NEXTHDR_AUTH:
  case NEXTHDR_DEST:
    if (skb_load_bytes(skb, l3_off + len, &opthdr, sizeof(opthdr)) < 0)
      return DROP_INVALID;
    nh = opthdr.nexthdr;
    if (nh == NEXTHDR_AUTH)
      len += ipv6_authlen(&opthdr);
    else
      len += ipv6_optlen(&opthdr);
}
\end{verbatim}
Another possibility is to use tail calls by calling into the same program again and using a
BPF_MAP_TYPE_PERCPU_ARRAY map for having a local scratch space. While being dynamic, this
form of looping however is limited to a maximum of 32 iterations.

In future, BPF may have some native, but limited form of implementing loops.

7. Partitioning programs with tail calls.

Tail calls provide the flexibility to atomically alter program behavior during runtime by jumping from one
BPF program into another. In order to select the next program, tail calls make use of program array maps
(BPF_MAP_TYPE_PROG_ARRAY), and pass the map as well as the index to the next program to jump
to. There is no return to the old program after the jump has been performed, and in case there was no
program present at the given map index, then execution continues on the original program.

For example, this can be used to implement various stages of a parser, where such stages could be updated
with new parsing features during runtime.

Another use case are event notifications, for example, Cilium can opt in packet drop notifications during
runtime, where the skb_event_output() call is located inside the tail called program. Thus, during
normal operations, the fall-through path will always be executed unless a program is added to the related
map index, where the program then prepares the meta data and triggers the event notification to a user
space daemon.

Program array maps are quite flexible, such that also individual actions can be implemented for programs
located in each map index. For example, the root program attached to XDP or tc could perform an initial
tail call to index 0 of the program array map, performing traffic sampling, then jumping to index 1 of the
program array map, where firewalling policy is applied and the packet either dropped or further processed
in index 2 of the program array map, where it is mangled and sent out of an interface again. Jumps in the
program array map can, of course, be arbitrary. The kernel will eventually execute the fall-through path
when the maximum tail call limit has been reached.

Minimal example extract of using tail calls:

```c
break;
default:
    *nexthdr = nh;
return len;
}
}
```
When loading this toy program, tc will create the program array and pin it to the BPF file system in the global namespace under jmp_map. Also, the BPF ELF loader in iproute2 will also recognize sections that are marked as __section_tail(). The provided id in struct bpf_elf_map will be matched against the id marker in the __section_tail(), that is, JMP_MAP_ID, and the program therefore loaded at the user specified program array map index, which is 0 in this example. As a result, all provided tail call sections will be populated by the iproute2 loader to the corresponding maps. This mechanism is not specific to tc, but can be applied with any other BPF program type that iproute2 supports (such as XDP, lwt).

The pinned map can be retrieved by a user space applications (e.g., Cilium daemon), but also by tc itself in order to update the map with new programs. Updates happen atomically, the initial entry programs that are triggered first from the various subsystems are also updated atomically.

Example for tc to perform tail call map updates:

```
# tc exec bpf graft m:globals/jmp_map key 0 obj new.o sec foo
```

In case iproute2 would update the pinned program array, the graft command can be used. By pointing it to globals/jmp_map, tc will update the map at index / key 0 with a new program residing in the object file new.o under section foo.

8. Limited stack space of 512 bytes.

Stack space in BPF programs is very limited, namely to 512 bytes, which needs to be taken into
careful consideration when implementing them in C. However, as mentioned earlier in point 3., a
BPF_MAP_TYPE_Percep_ARRAY map with a single entry can be used in order to enlarge scratch
buffer space.

**iproute2**

There are various front ends for loading BPF programs into the kernel such as bcc, perf, iproute2 and others. The Linux kernel source tree also provides a user space library under tools/lib/bpf/, which is mainly used and driven by perf for loading BPF tracing programs into the kernel. However, the library itself is generic and not limited to perf only. bcc is a toolkit that provides many useful BPF programs mainly for tracing that are loaded ad-hoc through a Python interface embedding the BPF C code. Syntax and semantics for implementing BPF programs slightly differ among front ends in general, though. Additionally, there are also BPF samples in the kernel source tree (samples/bpf/) that parse the generated object files and load the code directly through the system call interface.

This and previous sections mainly focus on the iproute2 suite’s BPF front end for loading networking programs of XDP, tc or lwt type, since Cilium’s programs are implemented against this BPF loader. In future, Cilium will ship with a native BPF loader, but programs will still be compatible to be loaded through iproute2 suite in order to facilitate development and debugging.

All BPF program types supported by iproute2 share the same BPF loader logic due to having a common loader back end implemented as a library (lib/bpf.c in iproute2 source tree).

The previous section on LLVM also covered some iproute2 parts related to writing BPF C programs, and later sections in this document are related to tc and XDP specific aspects when writing programs. Therefore, this section will rather focus on usage examples for loading object files with iproute2 as well as some of the generic mechanics of the loader. It does not try to provide a complete coverage of all details, but enough for getting started.

1. **Loading of XDP BPF object files.**

   Given an BPF object file prog.o has been compiled for XDP, it can be loaded through ip to a XDP-supported netdevice called em1 with the following command:

   ```
   # ip link set dev em1 xdp obj prog.o
   ```

   The above command assumes that the program code resides in the default section which is called prog in XDP case. Should this not be the case, and the section named differently, for example, foobar, then the program needs to be loaded as:

   ```
   # ip link set dev em1 xdp obj prog.o sec foobar
   ```

   By default, ip will throw an error in case a XDP program is already attached to the networking interface, thus that it will not be overridden by accident. In order to replace the currently running XDP program with a new one, the -force option must be used:

   ```
   # ip -force link set dev em1 xdp obj prog.o
   ```

   Most XDP-enabled drivers today support an atomic replacement of the existing program with a new one without traffic interruption. There is always only a single program attached to an XDP-enabled driver due to performance reasons, hence a chain of programs is not supported. However, as described in the previous section, partitioning of programs can be performed through tail calls to achieve a similar use-case when necessary.

   The ip link command will display an xdp flag if the interface has an XDP program attached. ip link | grep xdp can thus be used to find all interfaces that have XDP running. Further introspection facilities will be provided through the detailed view with ip -d link once the kernel API gains support for dumping additional attributes.
In order to remove the existing XDP program from the interface, the following command must be issued:

```
# ip link set dev em1 xdp off
```

2. Loading of tc BPF object files.

Given an BPF object file `prog.o` has been compiled for tc, it can be loaded through the `tc` command to a netdevice. Unlike XDP, there is no driver dependency for supporting attaching BPF programs to the device. Here, the netdevice is called `em1`, and with the following command the program can be attached to the networking ingress path of `em1`:

```
# tc qdisc add dev em1 clsact
# tc filter add dev em1 ingress bpf da obj prog.o
```

The first step is to set up a `clsact` qdisc (Linux queueing discipline). `clsact` is a dummy qdisc similar to the `ingress` qdisc, which can only hold classifier and actions, but does not perform actual queueing. It is needed in order to attach the `bpf` classifier. The `clsact` qdisc provides two special hooks called `ingress` and `egress`, where the classifier can be attached to. Both `ingress` and `egress` hooks are located at central receive and transmit locations in the networking data path, where every packet on the device passes through. The `ingress` hook is called from `__netif_receive_skb_core() -> sch_handle_ingress()` in the kernel and the `egress` hook from `__dev_queue_xmit() -> sch_handle_egress()`.

The equivalent for attaching the program to the `egress` hook looks as follows:

```
# tc filter add dev em1 egress bpf da obj prog.o
```

The `clsact` qdisc is processed lockless from `ingress` and `egress` direction and can also be attached to virtual, queue-less devices such as `veth` devices connecting containers.

Next to the hook, the `tc filter` command selects `bpf` to be used in da (direct-action) mode. da mode is recommended and should always be specified. It basically means that the `bpf` classifier does not need to call into external `tc` action modules, which are not necessary for `bpf` anyway, since all packet mangling, forwarding or other kind of actions can already be performed inside the single BPF program that is to be attached, and is therefore significantly faster.

At this point, the program has been attached and is executed once packets traverse the device. Like in XDP, should the default section name not be used, then it can be specified during load, for example, in case of section `foobar`:

```
# tc filter add dev em1 egress bpf da obj prog.o sec foobar
```

`iproute2`'s BPF loader allows for using the same command line syntax across program types, hence the `obj prog.o sec foobar` is the same syntax as with XDP mentioned earlier.

The attached programs can be listed through the following commands:

```
# tc filter show dev em1 ingress
filter protocol all pref 49152 bpf
filter protocol all pref 49152 bpf handle 0x1 prog.o:[ingress] direct-action_
  → tag c5f7825e5dac396f

# tc filter show dev em1 egress
filter protocol all pref 49152 bpf
filter protocol all pref 49152 bpf handle 0x1 prog.o:[egress] direct-action_
  → tag b2fd5adc0f262714
```

The output of `prog.o:[ingress]` tells that program section `ingress` was loaded from the file `prog.o`, and `bpf` operates in `direct-action` mode. The program tags are appended for each, which
denotes a hash over the instruction stream that can be used for debugging / introspection.

tc can attach more than just a single BPF program, it provides various other classifiers that can be chained together. However, attaching a single BPF program is fully sufficient since all packet operations can be contained in the program itself thanks to da (direct-action) mode. For optimal performance and flexibility, this is the recommended usage.

In the above show command, tc also displays pref 49152 and handle 0x1 next to the BPF related output. Both are auto-generated in case they are not explicitly provided through the command line. pref denotes a priority number, such that in case multiple classifiers are attached, they will be executed based on ascending priority, and handle represents an identifier in case multiple instances of the same classifier have been loaded under the same pref. Since in case of BPF, a single program is fully sufficient, pref and handle can typically be ignored.

Only in the case where it is planned to atomically replace the attached BPF programs, it would be recommended to explicitly specify pref and handle a-priori on initial load, such that they do not have to be queried at a later point in time for the replace operation. Thus, creation becomes:

```bash
# tc filter add dev em1 ingress pref 1 handle 1 bpf da obj prog.o sec foobar
# tc filter show dev em1 ingress
filter protocol all pref 1 bpf
filter protocol all pref 1 bpf handle 0x1 prog.o:[foobar] direct-action tag
  →c5f7825e5dac396f
```

And for the atomic replacement, the following can be issued for updating the existing program at ingress hook with the new BPF program from the file prog.o in section foobar:

```bash
# tc filter replace dev em1 ingress pref 1 handle 1 bpf da obj prog.o sec
  →foobar
```

Last but not least, in order to remove all attached programs from the ingress respectively egress hook, the following can be used:

```bash
# tc filter del dev em1 ingress
# tc filter del dev em1 egress
```

For removing the entire clsact qdisc from the netdevice, which implicitly also removes all attached programs from the ingress and egress hooks, the below command is provided:

```bash
# tc qdisc del dev em1 clsact
```

These two workflows are the basic operations to load XDP BPF respectively tc BPF programs with iproute2.

There are various other advanced options for the BPF loader that apply both to XDP and tc, some of them are listed here. In the examples only XDP is presented for simplicity.

### 1. Verbose log output even on success.

The option verb can be appended for loading programs in order to dump the verifier log, even if no error occurred:

```bash
# ip link set dev em1 xdp obj xdp.o verb
```

Prog section ’prog’ loaded (5)!
 - Type: 6
 - Instructions: 2 (0 over limit)
 - License: GPL
Verifier analysis:
0: (b7) r0 = 1
1: (95) exit
processed 2 insns

2. Load program that is already pinned in BPF file system.

Instead of loading a program from an object file, iproute2 can also retrieve the program from the BPF file system in case some external entity pinned it there and attach it to the device:

```
# ip link set dev em1 xdp pinned /sys/fs/bpf/prog
```

iproute2 can also use the short form that is relative to the detected mount point of the BPF file system:

```
# ip link set dev em1 xdp pinned m:prog
```

When loading BPF programs, iproute2 will automatically detect the mounted file system instance in order to perform pinning of nodes. In case no mounted BPF file system instance was found, then tc will automatically mount it to the default location under `/sys/fs/bpf/`.

In case an instance was already found, then it will be used and no additional mount will be performed:

```
# mkdir /var/run/bpf
# mount --bind /var/run/bpf /var/run/bpf
# mount -t bpf bpf /var/run/bpf
# tc filter add dev em1 ingress bpf da obj tc.o sec prog
# tree /var/run/bpf
/var/run/bpf
|-- ip -> /run/bpf/tc/
| `-- tc
|   |-- globals
|       `-- jmp_map
|-- xdp -> /run/bpf/tc/
4 directories, 1 file
```

By default tc will create an initial directory structure as shown above, where all subsystem users will point to the same location through symbolic links for the `globals` namespace, such that pinned BPF maps can be reused among various BPF program types in iproute2. In case the file system instance was mounted already and an existing structure exists already, then tc will not override it. This could be the case for separating `lwt`, `tc` and `xdp` maps in order to not share `globals` among all.

As briefly covered in the previous LLVM section, iproute2 will install a header file upon installation that can be included through the standard include path by BPF programs:

```
#include <iproute2/bpf_elf.h>
```

The header file’s purpose is to provide an API for maps and default section names used by programs. It’s a stable contract between iproute2 and BPF programs.

The map definition for iproute2 is `struct bpf_elf_map`. Its members have been covered earlier in the LLVM section of this document.

When parsing the BPF object file, the iproute2 loader will walk through all ELF sections. It initially fetches ancillary sections like `maps` and `license`. For `maps`, the `struct bpf_elf_map` array will be checked for validity and whenever needed, compatibility workarounds are performed. Subsequently all maps are created with the user provided information, either retrieved as a pinned object, or newly created and then pinned into the BPF file system. Next the
loader will handle all program sections that contain ELF relocation entries for maps, meaning that BPF instructions
that load map file descriptors into registers are rewritten such that the corresponding map file descriptors are encoded
into the instructions immediate value, so that the kernel can later on convert them into map kernel pointers. After that
all the programs themselves are created through the BPF system call, and tail called maps, if present, updated with the
program’s file descriptors.

**BPF sysctls**

The Linux kernel provides few sysctls that are BPF related and covered in this section.

- `/proc/sys/net/core/bpf_jit_enable`: Enables or disables the BPF JIT compiler.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disable the JIT and use only interpreter (kernel’s default value)</td>
</tr>
<tr>
<td>1</td>
<td>Enable the JIT compiler</td>
</tr>
<tr>
<td>2</td>
<td>Enable the JIT and emit debugging traces to the kernel log</td>
</tr>
</tbody>
</table>

As described in subsequent sections, `bpf_jit_disasm` tool can be used to process debugging traces when
the JIT compiler is set to debugging mode (option 2).

- `/proc/sys/net/core/bpf_jit_harden`: Enables or disables BPF JIT hardening. Note that enabling
hardening trades off performance, but can mitigate JIT spraying by blinding out the BPF program’s immediate
values. For programs processed through the interpreter, blinding of immediate values is not needed / performed.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disable JIT hardening (kernel’s default value)</td>
</tr>
<tr>
<td>1</td>
<td>Enable JIT hardening for unprivileged users only</td>
</tr>
<tr>
<td>2</td>
<td>Enable JIT hardening for all users</td>
</tr>
</tbody>
</table>

- `/proc/sys/net/core/bpf_jit_kallsyms`: Enables or disables export of JITed programs as kernel
symbols to `/proc/kallsyms` such that they can be used together with `perf` tooling as well as making these
addresses aware to the kernel for stack unwinding, for example, used in dumping stack traces. The symbol
names contain the BPF program tag (`bpf_prog_<tag>`). If `bpf_jit_harden` is enabled, then this feature
is disabled.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disable JIT kallsyms export (kernel’s default value)</td>
</tr>
<tr>
<td>1</td>
<td>Enable JIT kallsyms export for privileged users only</td>
</tr>
</tbody>
</table>

**Kernel Testing**

The Linux kernel ships an BPF selftest suite, which can be found in the kernel source tree under `tools/testing/selftests/bpf/`.

```
$ cd tools/testing/selftests/bpf/
$ make
# make run_tests
```

The test suite contains test cases against the BPF verifier, program tags, various tests against the BPF map interface
and map types. It contains various runtime tests from C code for checking LLVM back end, and eBPF as well as cBPF
asm code that is run in the kernel for testing the interpreter and JITs.
JIT Debugging

For JIT developers performing audits or writing extensions, each compile run can output the generated JIT image into the kernel log through:

```
# echo 2 > /proc/sys/net/core/bpf_jit_enable
```

Whenever a new BPF program is loaded, the JIT compiler will dump the output which can then be inspected with `dmesg`, for example:

```
[ 3389.935842] flen=6 proglen=70 pass=3 image=fffffffffa0069c8f from=tcpdump pid=20583
[ 3389.935847] JIT code: 00000000: 55 48 89 e5 48 83 ec 60 48 89 5d f8 44 8b 4f 68
[ 3389.935849] JIT code: 00000010: 44 2b 4f 6c 44 8b 87 d8 00 00 00 be 0c 00 00 00
[ 3389.935850] JIT code: 00000020: e8 1d 94 ff e0 3d 00 08 00 00 75 16 be 17 00 00
[ 3389.935851] JIT code: 00000030: 0e e8 28 94 ff e0 83 f8 01 75 07 b8 ff ff 00 00
[ 3389.935852] JIT code: 00000040: eb 02 31 c0 c9 c3
```

flen is the length of the BPF program (here, 6 BPF instructions), and proglen tells the number of bytes that were generated by the JIT for the opcode image (here, 70 bytes in size). pass means that the image was generated in 3 compiler passes, for example, x86_64 can have various optimization passes to further reduce the image size when possible. image contains the address of the generated JIT image, from and pid the user space application name and PID respectively, that triggered the compilation process. The dump output for eBPF and cBPF JITs is the same format.

In the kernel tree under tools/net/, there is a tool called `bpf_jit_disasm`. It reads out the latest dump and prints the disassembly for further inspection:

```
# ./bpf_jit_disasm
70 bytes emitted from JIT compiler (pass:3, flen:6)
fffffffffa0069c8f + <x>:
  0:  push  %rbp
  1:  mov   %rsp,%rbp
  4:  sub   $0x60,%rsp
  8:  mov   %rbx,-0x8(%rbp)
 c:  mov   0x68(%rdi),%r9d
10:  sub   0x6c(%rdi),%r9d
14:  mov   0xd8(%rdi),%r8
1b:  mov   $0xc,%esi
20:  callq 0xfffffffff0ff9442
25:  cmp   $0x800,%eax
2a:  jne   0x0000000000000042
2c:  mov   $0x17,%esi
31:  callq 0xfffffffff0ff945e
36:  cmp   $0x1,%eax
39:  jne   0x0000000000000042
3b:  mov   $0xffff,%eax
40:  jmp   0x0000000000000044
42:  xor   %eax,%eax
44:  leaveq
45:  retq
```

Alternatively, the tool can also dump related opcodes along with the disassembly.

```
# ./bpf_jit_disasm -o
70 bytes emitted from JIT compiler (pass:3, flen:6)
fffffffffa0069c8f + <x>:
  0:  push  %rbp
  1:  mov   %rsp,%rbp
  4:  sub   $0x60,%rsp
  8:  mov   %rbx,-0x8(%rbp)
 c:  mov   0x68(%rdi),%r9d
10:  sub   0x6c(%rdi),%r9d
14:  mov   0xd8(%rdi),%r8
1b:  mov   $0xc,%esi
20:  callq 0xfffffffff0ff9442
25:  cmp   $0x800,%eax
2a:  jne   0x0000000000000042
2c:  mov   $0x17,%esi
31:  callq 0xfffffffff0ff945e
36:  cmp   $0x1,%eax
39:  jne   0x0000000000000042
3b:  mov   $0xffff,%eax
40:  jmp   0x0000000000000044
42:  xor   %eax,%eax
44:  leaveq
45:  retq
```
For performance analysis of JITed BPF programs, perf can be used as usual. As a prerequisite, JITed programs need to be exported through kallsyms infrastructure.

```
# echo 1 > /proc/sys/net/core/bpf_jit_enable
# echo 1 > /proc/sys/net/core/bpf_jit_kallsyms
```

Enabling or disabling bpf_jit_kallsyms does not require a reload of the related BPF programs. Next, a small workflow example is provided for profiling BPF programs. A crafted tc BPF program is used for demonstration purposes, where perf records a failed allocation inside bpf_clone_redirect() helper. Due to the use of direct write, bpf_try_make_head_writable() failed that would then release the cloned skb again and return with an error message. perf thus records all kfree_skb events.

```
# tc qdisc add dev em1 clsact
# tc filter add dev em1 ingress bpf da obj prog.o sec main
# tc filter show dev em1 ingress
filter protocol all pref 49152 bpf
filter protocol all pref 49152 bpf handle 0xl prog.o:[main] direct-action tag
 8227addf251b7543
```
The stack trace recorded by `perf` will then show the `bpf_prog_8227addf251b7543()` symbol as part of the call trace, meaning the BPF program with the tag `8227addf251b7543` was related to the `kfree_skb` event, and such program was attached to netdevice `em1` on the ingress hook as shown by `tc`.

## Introspection

The Linux kernel provides various tracepoints around BPF and XDP that can be used for additional introspection, for example, to trace interactions of user space programs with the bpf system call.

### Tracepoints for BPF:

```bash
# perf list | grep bpf:
  bpf:bpf_map_create [Tracepoint event]
  bpf:bpf_map_delete_elem [Tracepoint event]
  bpf:bpf_map_lookup_elem [Tracepoint event]
  bpf:bpf_map_next_key [Tracepoint event]
  bpf:bpf_map_update_elem [Tracepoint event]
  bpf:bpf_obj_get_map [Tracepoint event]
  bpf:bpf_obj_get_prog [Tracepoint event]
  bpf:bpf_obj_pin_map [Tracepoint event]
  bpf:bpf_obj_pin_prog [Tracepoint event]
  bpf:bpf_prog_get_type [Tracepoint event]
  bpf:bpf_prog_load [Tracepoint event]
  bpf:bpf_prog_put_rcu [Tracepoint event]
```

Example usage with `perf` (alternatively to `sleep` example used here, a specific application like `tc` could be used here instead, of course):

```bash
# perf record -a -e bpf:* sleep 10
# perf script
```
For the BPF programs, their individual program tag is displayed.

For debugging, XDP also has a tracepoint that is triggered when exceptions are raised:

```bash
# perf list | grep xdp:
xdp:xdp_exception  [Tracepoint event]
```

Exceptions are triggered in the following scenarios:

- The BPF program returned an invalid / unknown XDP action code.
- The BPF program returned with XDP_ABORTED indicating a non-graceful exit.
- The BPF program returned with XDP_TX, but there was an error on transmit, for example, due to the port not being up, due to the transmit ring being full, due to allocation failures, etc.

Both tracepoint classes can also be inspected with a BPF program itself that is attached to one or more tracepoints, collecting further information in a map or punting such events to a user space collector through the `bpf_perf_event_output()` helper, for example.

**Miscellaneous**

BPF programs and maps are memory accounted against `RLIMIT_MEMLOCK` similar to `perf`. The currently available size in unit of system pages that may be locked into memory can be inspected through `ulimit -l`. The `setrlimit` system call man page provides further details.

The default limit is usually insufficient to load more complex programs or larger BPF maps, such that the BPF system call will return with `errno` of `EPERM`. In such situations a workaround with `ulimit -l unlimited` or with a sufficiently large limit could be performed. The `RLIMIT_MEMLOCK` is mainly enforcing limits for unprivileged users. Depending on the setup, setting a higher limit for privileged users is often acceptable.

**tc (traffic control)**

TODO

**XDP**

TODO
References

Projects using BPF

The following projects are making use of BPF. This list is probably not complete, feel free to open pull requests to complete the list.

- BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more (https://github.com/iovisor/bcc)
- Cilium (https://github.com/cilium/cilium)
- ply - a dynamic tracer for Linux (https://wkz.github.io/ply)
- Go bindings for creating BPF programs (https://github.com/iovisor/gobpf)
- Suricata IDS (https://suricata-ids.org)

Talks & Publications

The following list includes publications and talks related to BPF and XDP:

Further Reading

- Dive into BPF: a list of reading material, Quentin Monnet (https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/)
CHAPTER 6

API Reference

GET /healthz
Get health of Cilium daemon
Returns health and status information of the Cilium daemon and related components such as the local container runtime, connected datastore, Kubernetes integration.

Status Codes
• 200 OK – Success

GET /config
Get configuration of Cilium daemon
Returns the configuration of the Cilium daemon.

Status Codes
• 200 OK – Success

PATCH /config
Modify daemon configuration
Updates the daemon configuration by applying the provided ConfigurationMap and regenerates & compiles all required datapath components.

Status Codes
• 200 OK – Success
• 400 Bad Request – Bad configuration parameters
• 500 Internal Server Error – Recompilation failed

GET /endpoint/{id}
Get endpoint by endpoint ID
Returns endpoint information

Parameters
• **id (string)** – String describing an endpoint with the format *prefix:id*. If no prefix is specified, a prefix of *cilium-local:* is assumed. Not all endpoints will be addressable by all endpoint ID prefixes with the exception of the local Cilium UUID which is assigned to all endpoints.

**Supported endpoint id prefixes:**

- cilium-local: Local Cilium endpoint UUID, e.g. cilium-local:3389595
- cilium-global: Global Cilium endpoint UUID, e.g. cilium-global:cluster1:nodeX:452343
- container-id: Container runtime ID, e.g. container-id:22222
- docker-net-endpoint: Docker libnetwork endpoint ID, e.g. docker-net-endpoint:4444

**Status Codes**

- 200 OK – Success
- 400 Bad Request – Invalid endpoint ID format for specified type
- 404 Not Found – Endpoint not found

**PUT /endpoint/{id}**

Create endpoint

Updates an existing endpoint

**Parameters**

- **id (string)** – String describing an endpoint with the format *prefix:id*. If no prefix is specified, a prefix of *cilium-local:* is assumed. Not all endpoints will be addressable by all endpoint ID prefixes with the exception of the local Cilium UUID which is assigned to all endpoints.

**Supported endpoint id prefixes:**

- cilium-local: Local Cilium endpoint UUID, e.g. cilium-local:3389595
- cilium-global: Global Cilium endpoint UUID, e.g. cilium-global:cluster1:nodeX:452343
- container-id: Container runtime ID, e.g. container-id:22222
- docker-net-endpoint: Docker libnetwork endpoint ID, e.g. docker-net-endpoint:4444

**Status Codes**

- 201 Created – Created
- 400 Bad Request – Invalid endpoint in request
- 409 Conflict – Endpoint already exists
- 500 Internal Server Error – Endpoint creation failed

**PATCH /endpoint/{id}**

Modify existing endpoint

Applies the endpoint change request to an existing endpoint

**Parameters**

- **id (string)** – String describing an endpoint with the format *prefix:id*. If no prefix is specified, a prefix of *cilium-local:* is assumed. Not all endpoints will be addressable by all
endpoint ID prefixes with the exception of the local Cilium UUID which is assigned to all endpoints.

**Supported endpoint id prefixes:**

- **cilium-local**: Local Cilium endpoint UUID, e.g. `cilium-local:3389595`
- **cilium-global**: Global Cilium endpoint UUID, e.g. `cilium-global:cluster1:nodeX:452343`
- **container-id**: Container runtime ID, e.g. `container-id:22222`
- **docker-net-endpoint**: Docker libnetwork endpoint ID, e.g. `docker-net-endpoint:4444`

**Status Codes**

- **200 OK** – Success
- **400 Bad Request** – Invalid modify endpoint request
- **404 Not Found** – Endpoint does not exist
- **500 Internal Server Error** – Endpoint update failed

**DELETE /endpoint/{id}**

Delete endpoint

Deletes the endpoint specified by the ID. Deletion is imminent and atomic, if the deletion request is valid and the endpoint exists, deletion will occur even if errors are encountered in the process. If errors have been encountered, the code 202 will be returned, otherwise 200 on success.

All resources associated with the endpoint will be freed and the workload represented by the endpoint will be disconnected. It will no longer be able to initiate or receive communications of any sort.

**Parameters**

- **id** *(string)* – String describing an endpoint with the format `/prefix:/id`. If no prefix is specified, a prefix of `cilium-local:` is assumed. Not all endpoints will be addressable by all endpoint ID prefixes with the exception of the local Cilium UUID which is assigned to all endpoints.

**Supported endpoint id prefixes:**

- **cilium-local**: Local Cilium endpoint UUID, e.g. `cilium-local:3389595`
- **cilium-global**: Global Cilium endpoint UUID, e.g. `cilium-global:cluster1:nodeX:452343`
- **container-id**: Container runtime ID, e.g. `container-id:22222`
- **docker-net-endpoint**: Docker libnetwork endpoint ID, e.g. `docker-net-endpoint:4444`

**Status Codes**

- **200 OK** – Success
- **206 Partial Content** – Deleted with a number of errors encountered
- **400 Bad Request** – Invalid endpoint ID format for specified type. Details in error message
- **404 Not Found** – Endpoint not found

**GET /endpoint**

Get list of all endpoints

Returns an array of all local endpoints.

**Status Codes**
• 200 OK – Success

**GET /endpoint/{id}/config**

*Retrieve endpoint configuration*

Retrieves the configuration of the specified endpoint.

**Parameters**

- **id** (*string*) – String describing an endpoint with the format *prefix:*id. If no prefix is specified, a prefix of *cilium-local:* is assumed. Not all endpoints will be addressable by all endpoint ID prefixes with the exception of the local Cilium UUID which is assigned to all endpoints.

**Supported endpoint id prefixes:**

- *cilium-local:* Local Cilium endpoint UUID, e.g. cilium-local:3389595
- *cilium-global:* Global Cilium endpoint UUID, e.g. cilium-global:cluster1:nodeX:452343
- *container-id:* Container runtime ID, e.g. container-id:22222
- *docker-net-endpoint:* Docker libnetwork endpoint ID, e.g. docker-net-endpoint:4444

**Status Codes**

- 200 OK – Success
- 404 Not Found – Endpoint not found

**PATCH /endpoint/{id}/config**

*Modify mutable endpoint configuration*

Update the configuration of an existing endpoint and regenerates & recompiles the corresponding programs automatically.

**Parameters**

- **id** (*string*) – String describing an endpoint with the format *prefix:*id. If no prefix is specified, a prefix of *cilium-local:* is assumed. Not all endpoints will be addressable by all endpoint ID prefixes with the exception of the local Cilium UUID which is assigned to all endpoints.

**Supported endpoint id prefixes:**

- *cilium-local:* Local Cilium endpoint UUID, e.g. cilium-local:3389595
- *cilium-global:* Global Cilium endpoint UUID, e.g. cilium-global:cluster1:nodeX:452343
- *container-id:* Container runtime ID, e.g. container-id:22222
- *docker-net-endpoint:* Docker libnetwork endpoint ID, e.g. docker-net-endpoint:4444

**Status Codes**

- 200 OK – Success
- 400 Bad Request – Invalid configuration request
- 404 Not Found – Endpoint not found

**GET /endpoint/{id}/labels**

Retrieves the list of labels associated with an endpoint.
Parameters

• id (string) – String describing an endpoint with the format [prefix:]id. If no prefix is specified, a prefix of cilium-local: is assumed. Not all endpoints will be addressable by all endpoint ID prefixes with the exception of the local Cilium UUID which is assigned to all endpoints.

Supported endpoint id prefixes:

– cilium-local: Local Cilium endpoint UUID, e.g. cilium-local:3389595
– cilium-global: Global Cilium endpoint UUID, e.g. cilium-global:cluster1:nodeX:452343
– container-id: Container runtime ID, e.g. container-id:22222
– docker-net-endpoint: Docker libnetwork endpoint ID, e.g. docker-net-endpoint:4444

Status Codes

• 200 OK – Success
• 404 Not Found – Endpoint not found

PUT /endpoint/(id)/labels
Modify label configuration of endpoint

Updates the list of labels associated with an endpoint by applying a label modifier structure to the label configuration of an endpoint.

The label configuration mutation is only executed as a whole, i.e. if any of the labels to be deleted are not either on the list of orchestration system labels, custom labels, or already disabled, then the request will fail. Labels to be added which already exist on either the orchestration list or custom list will be ignored.

Parameters

• id (string) – String describing an endpoint with the format [prefix:]id. If no prefix is specified, a prefix of cilium-local: is assumed. Not all endpoints will be addressable by all endpoint ID prefixes with the exception of the local Cilium UUID which is assigned to all endpoints.

Supported endpoint id prefixes:

– cilium-local: Local Cilium endpoint UUID, e.g. cilium-local:3389595
– cilium-global: Global Cilium endpoint UUID, e.g. cilium-global:cluster1:nodeX:452343
– container-id: Container runtime ID, e.g. container-id:22222
– docker-net-endpoint: Docker libnetwork endpoint ID, e.g. docker-net-endpoint:4444

Status Codes

• 200 OK – Success
• 404 Not Found – Endpoint not found
• 460 – Label to be deleted not found
• 500 Internal Server Error – Error while updating labels

GET /identity
Retrieve identity by labels

Status Codes

• 200 OK – Success
• 404 Not Found – Endpoint not found
- 200 OK – Success
- 404 Not Found – Identity not found
- 521 – Invalid identity format in storage

**GET /identity/{id}**
Retrieve identity

**Parameters**
- **id**(string) – Cluster wide unique identifier of a security identity.

**Status Codes**
- 200 OK – Success
- 400 Bad Request – Invalid identity provided
- 404 Not Found – Identity not found
- 521 – Invalid identity format in storage

**POST /ipam**
Allocate an IP address

**Query Parameters**
- **family**(string)

**Status Codes**
- 201 Created – Success
- 502 Bad Gateway – Allocation failure

**POST /ipam/{ip}**
Allocate an IP address

**Parameters**
- **ip**(string) – IP address

**Status Codes**
- 200 OK – Success
- 400 Bad Request – Invalid IP address
- 409 Conflict – IP already allocated
- 501 Not Implemented – Allocation for address family disabled

**DELETE /ipam/{ip}**
Release an allocated IP address

**Parameters**
- **ip**(string) – IP address

**Status Codes**
- 200 OK – Success
• 400 Bad Request – Invalid IP address
• 404 Not Found – IP address not found
• 500 Internal Server Error – Address release failure
• 501 Not Implemented – Allocation for address family disabled

GET /policy
Retrieve entire policy tree
Returns the entire policy tree with all children.

Status Codes
• 200 OK – Success

GET /policy/{path}
Retrieve policy subtree
Returns the policy node at path with all its children

Parameters
• path (string) – Path to policy node

Status Codes
• 200 OK – Success
• 400 Bad Request – Invalid policy path
• 404 Not Found – Policy tree not found

PUT /policy/{path}
Create or update a policy (sub)tree

Parameters
• path (string) – Path to policy node

Status Codes
• 200 OK – Success
• 400 Bad Request – Invalid policy
• 460 – Invalid path
• 500 Internal Server Error – Policy import failed

DELETE /policy/{path}
Delete a policy (sub)tree

Parameters
• path (string) – Path to policy node

Status Codes
• 204 No Content – Success
• 400 Bad Request – Invalid request
• 404 Not Found – Policy tree not found
• 500 Internal Server Error – Error while deleting policy

GET /policy/resolve
Resolve policy for an identity context
Status Codes

• 200 OK – Success

GET /service
Retrieve list of all services

Status Codes

• 200 OK – Success

GET /service/{id}
Retrieve configuration of a service

Parameters

• id (integer) – ID of service

Status Codes

• 200 OK – Success
• 404 Not Found – Service not found

PUT /service/{id}
Create or update service

Parameters

• id (integer) – ID of service

Status Codes

• 200 OK – Updated
• 201 Created – Created
• 460 – Invalid frontend in service configuration
• 461 – Invalid backend in service configuration
• 500 Internal Server Error – Error while creating service

DELETE /service/{id}
Delete a service

Parameters

• id (integer) – ID of service

Status Codes

• 200 OK – Success
• 404 Not Found – Service not found
• 500 Internal Server Error – Service deletion failed
We’re happy you’re interested in contributing to the Cilium project. This guide will help you make sure you have an environment capable of testing changes to the Cilium source code, and that you understand the workflow of getting these changes reviewed and merged upstream.

### Setting up the a development environment

#### Developer requirements

You need to have the following tools available in order to effectively contribute to Cilium:

- `git`
- `go-swagger` `go get -u github.com/go-swagger/go-swagger/cmd/swagger`
- `go-bindata` `go get -u github.com/jteeuwen/go-bindata/...`

#### Testsuite

Please test all changes by running the testsuites. You have several options, you can either run the vagrant provisioner `testsuite` as follows:

```bash
$ vagrant provision --provision-with testsuite
```

or you can `vagrant ssh` into the machine and then run the tests yourself:

```bash
$ sudo make runtime-tests
```
Submitting a pull request

Contributions may be submitted in the form of pull requests against the github repository at: [https://github.com/cilium/cilium]

Before hitting the submit button, please make sure that the following requirements have been met:

- The pull request and all corresponding commits have been equipped with a well written commit message which explains the reasoning and details of the change.
- You have added unit and/or runtime tests where feasible.
- You have tested the changes and checked for regressions by running the existing testsuite against your changes. See the “Testsuite” section for additional details.
- You have signed off on your commits, see the section “Developer’s Certificate of Origin” for more details.

Release Process

Cilium schedules a major release every 3 months. Each major release is performed by incrementing the Y in the version format X.Y.0. The group of committers can decide to increment X instead to mark major milestones in which case Y is reset to 0.

The following steps are performed to publish a release:

1. The master branch is set to the version X.Y.90 at all times. This ensures that a development snapshot is considered more recent than a stable release at all times.
2. The committers can agree on a series of release candidates which will be tagged vX.Y-rcN in the master branch.
3. The committers declare the master branch ready for the release and fork the master branch into a release branch vX.Y+1.0.
4. The first commit in the release branch is to change the version to vX.Y+1.0.
5. The next commit goes into the master branch and sets the version to vX.Y+1.90 to ensure that the master branch will be considered more recent than any stable release of the major release that is about to be published.

Stable releases

The committers can nominate commits pushed to the master as stable release candidates in which case they will be backported to previous release branches. Upon necessity, stable releases are published with the version X.Y.Z+1.

Criteria for the inclusion into stable release branches are:

- Security relevant fixes
- Major bugfixes relevant to the correct operation of Cilium

Developer’s Certificate of Origin

To improve tracking of who did what, we’ve introduced a “sign-off” procedure.

The sign-off is a simple line at the end of the explanation for the commit, which certifies that you wrote it or otherwise have the right to pass it on as open-source work. The rules are pretty simple: if you can certify the below:
Developer Certificate of Origin
Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors.
1 Letterman Drive
Suite D4700
San Francisco, CA, 94129

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or

(b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or

(c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.

(d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

then you just add a line saying:

Signed-off-by: Random J Developer <random@developer.example.org>

Use your real name (sorry, no pseudonyms or anonymous contributions.)

Cilium Committer Grant/Revocation Policy

A Cilium committer is a participant in the project with the ability to commit code directly to the master repository. Commit access grants a broad ability to affect the progress of the project as presented by its most important artifact, the code and related resources that produce working binaries of Cilium. As such it represents a significant level of trust in an individual’s commitment to working with other committers and the community at large for the benefit of the project. It can not be granted lightly and, in the worst case, must be revocable if the trust placed in an individual was inappropriate.

This document suggests guidelines for granting and revoking commit access. It is intended to provide a framework for evaluation of such decisions without specifying deterministic rules that wouldn’t be sensitive to the nuance of specific situations. In the end the decision to grant or revoke committer privileges is a judgment call made by the existing set
Expectations for Developers with commit access

Pre-requisites

Be familiar with the Developer / Contributor Guide.

Review

Code (yours or others’) must be reviewed publicly (by you or others) before you push it to the repository. With one exception (see below), every change needs at least one review.

If one or more people know an area of code particularly well, code that affects that area should ordinarily get a review from one of them.

The riskier, more subtle, or more complicated the change, the more careful the review required. When a change needs careful review, use good judgment regarding the quality of reviews. If a change adds 1000 lines of new code, and a review posted 5 minutes later says just “Looks good,” then this is probably not a quality review.

(The size of a change is correlated with the amount of care needed in review, but it is not strictly tied to it. A search and replace across many files may not need much review, but one-line optimization changes can have widespread implications.)

Your own small changes to fix a recently broken build (“make”) or tests (“make check”), that you believe to be visible to a large number of developers, may be checked in without review. If you are not sure, ask for review.

Regularly review submitted code in areas where you have expertise. Consider reviewing other code as well.

Git conventions

If you apply a change (yours or another’s) then it is your responsibility to handle any resulting problems, especially broken builds and other regressions. If it is someone else’s change, then you can ask the original submitter to address it. Regardless, you need to ensure that the problem is fixed in a timely way. The definition of “timely” depends on the severity of the problem.

If a bug is present on master and other branches, fix it on master first, then backport the fix to other branches. Straight-forward backports do not require additional review (beyond that for the fix on master).

Feature development should be done only on master. Occasionally it makes sense to add a feature to the most recent release branch, before the first actual release of that branch. These should be handled in the same way as bug fixes, that is, first implemented on master and then backported.

Keep the authorship of a commit clear by maintaining a correct list of “Signed-off-by:”s. If a confusing situation comes up, as it occasionally does, bring it up in the development forums. If you explain the use of “Signed-off-by:” to a new developer, explain not just how but why, since the intended meaning of “Signed-off-by:” is more important than the syntax.

Use Reported-by: and Tested-by: tags in commit messages to indicate the source of a bug report.

Keep the AUTHORS file up to date.
Granting Commit Access

Granting commit access should be considered when a candidate has demonstrated the following in their interaction with the project:

- Contribution of significant new features through the patch submission process where:
  - Submissions are free of obvious critical defects
  - Submissions do not typically require many iterations of improvement to be accepted
  - Consistent participation in code review of other's patches, including existing committers, with comments consistent with the overall project standards
  - Assistance to those in the community who are less knowledgeable through active participation in project forums.
  - Plans for sustained contribution to the project compatible with the project's direction as viewed by current committers.
  - Commitment to meet the expectations described in the “Expectations of Developer’s with commit access”

The process to grant commit access to a candidate is simple:

- An existing committer nominates the candidate by sending an email to all existing committers with information substantiating the contributions of the candidate in the areas described above.
- All existing committers discuss the pros and cons of granting commit access to the candidate in the email thread.
- When the discussion has converged or a reasonable time has elapsed without discussion developing (e.g. a few business days) the nominator calls for a final decision on the candidate with a followup email to the thread.
- Each committer may vote yes, no, or abstain by replying to the email thread. A failure to reply is an implicit abstention.
- After votes from all existing committers have been collected or a reasonable time has elapsed for them to be provided (e.g. a couple of business days) the votes are evaluated. To be granted commit access the candidate must receive yes votes from a majority of the existing committers and zero no votes. Since a no vote is effectively a veto of the candidate it should be accompanied by a reason for the vote.
- The nominator summarizes the result of the vote in an email to all existing committers.
- If the vote to grant commit access passed, the candidate is contacted with an invitation to become a committer to the project which asks them to agree to the committer expectations documented on the project web site.
- If the candidate agrees access is granted by setting up commit access to the repos.

Revoking Commit Access

There are two situations in which commit access might be revoked.

The straightforward situation is a committer who is no longer active in the project and has no plans to become active in the near future. The process in this case is:

- Any time after a committer has been inactive for more than 6 months any other committer to the project may identify that committer as a candidate for revocation of commit access due to inactivity.
- The plans of revocation should be sent in a private email to the candidate.
- If the candidate for removal states plans to continue participating no action is taken and this process terminates.
- If the candidate replies they no longer require commit access then commit access is removed and a notification is sent to the candidate and all existing committers.
- If the candidate can not be reached within 1 week of the first attempting to contact this process continues.
• A message proposing removal of commit access is sent to the candidate and all other committers.
• If the candidate for removal states plans to continue participating no action is taken.
• If the candidate replies they no longer require commit access then their access is removed.
• If the candidate can not be reached within 2 months of the second attempting to contact them, access is removed.
• In any case, where access is removed, this fact is published through an email to all existing committers (including the candidate for removal).

The more difficult situation is a committer who is behaving in a manner that is viewed as detrimental to the future of the project by other committers. This is a delicate situation with the potential for the creation of division within the greater community and should be handled with care. The process in this case is:

• Discuss the behavior of concern with the individual privately and explain why you believe it is detrimental to the project. Stick to the facts and keep the email professional. Avoid personal attacks and the temptation to hypothesize about unknowable information such as the other’s motivations. Make it clear that you would prefer not to discuss the behavior more widely but will have to raise it with other contributors if it does not change. Ideally the behavior is eliminated and no further action is required. If not,
• Start an email thread with all committers, including the source of the behavior, describing the behavior and the reason it is detrimental to the project. The message should have the same tone as the private discussion and should generally repeat the same points covered in that discussion. The person whose behavior is being questioned should not be surprised by anything presented in this discussion. Ideally the wider discussion provides more perspective to all participants and the issue is resolved. If not,
• Start an email thread with all committers except the source of the detrimental behavior requesting a vote on revocation of commit rights. Cite the discussion among all committers and describe all the reasons why it was not resolved satisfactorily. This email should be carefully written with the knowledge that the reasoning it contains may be published to the larger community to justify the decision.
• Each committer may vote yes, no, or abstain by replying to the email thread. A failure to reply is an implicit abstention.
• After all votes have been collected or a reasonable time has elapsed for them to be provided (e.g. a couple of business days) the votes are evaluated. For the request to revoke commit access for the candidate to pass it must receive yes votes from two thirds of the existing committers.
• anyone that votes no must provide their reasoning, and
• if the proposal passes then counter-arguments for the reasoning in no votes should also be documented along with the initial reasons the revocation was proposed. Ideally there should be no new counter-arguments supplied in a no vote as all concerns should have surfaced in the discussion before the vote.
• The original person to propose revocation summarizes the result of the vote in an email to all existing committers excepting the candidate for removal.
• If the vote to revoke commit access passes, access is removed and the candidate for revocation is informed of that fact and the reasons for it as documented in the email requesting the revocation vote.
• Ideally the revoked committer peacefully leaves the community and no further action is required. However, there is a distinct possibility that he/she will try to generate support for his/her point of view within the larger community. In this case the reasoning for removing commit access as described in the request for a vote will be published to the community.

Changing the Policy

The process for changing the policy is:

• Propose the changes to the policy in an email to all current committers and request discussion.
• After an appropriate period of discussion (a few days) update the proposal based on feedback if required and resend it to all current committers with a request for a formal vote.

• After all votes have been collected or a reasonable time has elapsed for them to be provided (e.g. a couple of business days) the votes are evaluated. For the request to modify the policy to pass it must receive yes votes from two thirds of the existing committers.

**Template Emails**

**Nomination to Grant Commit Access**

I would like to nominate *[candidate]* for commit access. I believe *[he/she]* has met the conditions for commit access described in the committer grant policy on the project website in the following ways:

*[list of requirements & evidence]*

Please reply to all in this message thread with your comments and questions. If that discussion concludes favorably I will request a formal vote on the nomination in a few days.

**Vote to Grant Commit Access**

I nominated *[candidate]* for commit access on *[date]*. Having allowed sufficient time for discussion it’s now time to formally vote on the proposal.

Please reply to all in this thread with your vote of: YES, NO, or ABSTAIN. A failure to reply will be counted as an abstention. If you vote NO, by our policy you must include the reasons for that vote in your reply. The deadline for votes is *[date and time]*.

If a majority of committers vote YES and there are zero NO votes commit access will be granted.

**Vote Results for Grant of Commit Access**

The voting period for granting to commit access to *[candidate]* initiated at *[date and time]* is now closed with the following results:

- YES: *[count of yes votes]* (*[% of voters]*)
- NO: *[count of no votes]* (*[% of voters]*)
- ABSTAIN: *[count of abstentions]* (*[% of voters]*)

Based on these results commit access *[is/is NOT]* granted.
Invitation to Accepted Committer

Due to your sustained contributions to the Cilium project we would like to provide you with commit access to the project repository. Developers with commit access must agree to fulfill specific responsibilities described in the source repository:

/Documentation/commit-access.rst

Please let us know if you would like to accept commit access and if so that you agree to fulfill these responsibilities. Once we receive your response we'll set up access. We're looking forward continuing to work together to advance the Cilium project.

Proposal to Remove Commit Access for Inactivity

Committer *[candidate]* has been inactive for *[duration]*. I have attempted to privately contacted *[him/her]* and *[he/she]* could not be reached.

Based on this I would like to formally propose removal of commit access. If a response to this message documenting the reasons to retain commit access is not received by *[date]* access will be removed.

Notification of Commit Removal for Inactivity

Committer *[candidate]* has been inactive for *[duration]*. *[He/she]* *[stated no commit access is required/failed to respond]* to the formal proposal to remove access on *[date]*. Commit access has now been removed.

Proposal to Revoke Commit Access for Detrimental Behavior

I regret that I feel compelled to propose revocation of commit access for *[candidate]*. I have privately discussed with *[him/her]* the following reasons I believe *[his/her]* actions are detrimental to the project and we have failed to come to a mutual understanding:

*[List of reasons and supporting evidence]*

Please reply to all in this thread with your thoughts on this proposal. I plan to formally propose a vote on the proposal on or after *[date and time]*.

It is important to get all discussion points both for and against the proposal on the table during the discussion period prior to the vote. Please make it a high priority to respond to this proposal with your thoughts.
Vote to Revoke Commit Access

I nominated *[candidate]* for revocation of commit access on *[date]*. Having allowed sufficient time for discussion it's now time to formally vote on the proposal.

Please reply to all in this thread with your vote of: YES, NO, or ABSTAIN. A failure to reply will be counted as an abstention. If you vote NO, by our policy you must include the reasons for that vote in your reply. The deadline for votes is *[date and time]*.

If 2/3rds of committers vote YES commit access will be revoked.

The following reasons for revocation have been given in the original proposal or during discussion:

*[list of reasons to remove access]*

The following reasons for retaining access were discussed:

*[list of reasons to retain access]*

The counter-argument for each reason for retaining access is:

*[list of counter-arguments for retaining access]*

Vote Results for Revocation of Commit Access

The voting period for revoking the commit access of *[candidate]* initiated at *[date and time]* is now closed with the following results:

- YES: *[count of yes votes]* (% of voters)
- NO: *[count of no votes]* (% of voters)
- ABSTAIN: *[count of abstentions]* (% of voters)

Based on these results commit access *[is/is NOT]* revoked. The following reasons for retaining commit access were proposed in NO votes:

*[list of reasons]*

The counter-arguments for each of these reasons are:

*[list of counter-arguments]*

Notification of Commit Revocation for Detrimental Behavior

After private discussion with you and careful consideration of the situation, the other committers to the Cilium project have concluded that it is in the best interest of the project that your commit access to the project repositories be revoked and this has now occurred.

The reasons for this decision are:
While your goals and those of the project no longer appear to be aligned we greatly appreciate all the work you have done for the project and wish you continued success in your future work.
HTTP Routing Table

/config
GET /config, 63
PATCH /config, 63

/endpoint
GET /endpoint, 65
GET /endpoint/{id}, 63
GET /endpoint/{id}/config, 66
GET /endpoint/{id}/labels, 66
PUT /endpoint/{id}, 64
PUT /endpoint/{id}/labels, 67
DELETE /endpoint/{id}, 65
PATCH /endpoint/{id}, 64
PATCH /endpoint/{id}/config, 66

/healthz
GET /healthz, 63

/identity
GET /identity, 67
GET /identity/{id}, 68

/ipam
POST /ipam, 68
POST /ipam/{ip}, 68
DELETE /ipam/{ip}, 68

/policy
GET /policy, 69
GET /policy/resolve, 69
GET /policy/{path}, 69
PUT /policy/{path}, 69
DELETE /policy/{path}, 69

/service
GET /service, 70
GET /service/{id}, 70
PUT /service/{id}, 70
DELETE /service/{id}, 70