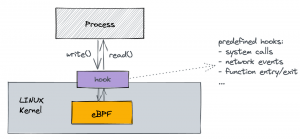
eBPF DOCUMENTAZIONE

* https://www.solo.io/blog/get-started-with-ebpf-using-bumblebee/

[eBPF](https://ebpf.io/what-is-ebpf) stands for extended Berkeley Packet Filter. The Linux kernel has been around for a long time, however, it is not easy to modify or extend the kernel unless you know how to patch it. If you are familiar with [Kubernetes’s custom resources](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/) or Envoy filters, you understand how important it is to build extensions based on your specific scenarios. What eBPF provides to the Linux kernel is the extensibility to enable developers to program the Linux kernel to quickly build intelligent or feature-rich functions based on their business needs. eBPF programs to the Linux kernel are similar to what web assembly modules are to Envoy. They allow developers to extend the kernel easily and run their eBPF code as sandboxed programs in the kernel without changing the kernel source code or loading kernel modules.



Options for building eBPF programs

Belowe there are the options explored while looking into building an eBPF program.

1. BPF Compiler Collection (BCC)

BCC is a toolkit for creating efficient kernel tracing and manipulation eBPF programs, which requires Linux 4.1 and above. While BCC is designed to make BPF programs easier to write, it requires writing the kernel instrumentation in C wrapped as a plain string into your user-space program in Python or Lua. There is both C code and python code in any eBPF python examples. If you are not a Linux user, it is hard to find an environment where you can compile and test your first eBPF program. It can be used a [Vagrantfile](https://github.com/booyaa/vagrant-bcctools/blob/master/Vagrantfile) to set up the environment to build and run an hello world eBPF program.

Once gotten the first hello world eBPF program compiled and running, it is useful to know how portable it is. Can the compiled binary be taken and run it on any kernel above 4.1? Or does it need to be compiled for each kernel version that you want to run? When your eBPF program is deployed and executed, BCC invokes its embedded Clang/LLVM, pulls in local kernel headers (which you have to make sure are installed on the system from the correct kernel-devel package), and performs compilation of the plain string on the fly on the Linux kernel. While this method tailors your BPF code to the specific kernel, it can contain major drawbacks such as fat library distributed with your application, heavy resource utilization, requiring kernel headers, and more. But there are ways to address these challenges as well.

1. Enter BPF CO-RE and libbpf

BPF CO-RE (Compile Once Run Everywhere) is targeted to solve the above portability problem so you only need to compile once without needing to compile it across each particular kernel version. [libbpf](https://github.com/libbpf), an alternate set of tools for building BPF applications, is introduced as a BPF program loader, which knows how to tailor a BPF program code to a particular kernel on the host. It resolves and matches all the BTF (BPF Type Format) types and fields, updates necessary offsets and other relocatable data as needed to ensure that the BPF program’s logic is correctly functioning for a specific kernel on the host. If everything checks out, you will get a BPF program for the kernel on the target host as if your program was specifically compiled for it.

With libbpf, you will write plain C code along with helper marco to eliminate the mundane parts. What is really neat is that what you write is what gets executed and you no longer need to wrap your BPF code as a plain string in Python or Lua. This approach keeps overhead to the minimum, eliminates heavy dependencies, and makes BPF more practical.

1. Simplify BPF program with BumbleBee

In a nutshell, [BumbleBee](https://bumblebee.io/) brings a Docker-like experience for eBPF to you. Through simple bee CLI commands, you can easily build, run, and distribute your eBPF programs as [OCI (Open Container Initiative) images](https://github.com/opencontainers/image-spec) and plug the images to your existing OCI image workflows for publishing and distribution. BumbleBee is built using libbpf and allows you to focus on writing your eBPF code while taking care of the user space components automatically for you, e.g. you can write eBPF probes with zero user space code. BumbleBee automatically detects and displays maps in your program that allow the user space and kernel space programs to share data. This is accomplished through the use of special BPF [conventions](https://github.com/solo-io/bumblebee/blob/main/docs/concepts.md#BPF-conventions) and keywords.

PROGRAMMING eBPF PROGRAM WITH BEE

* https://bumblebee.io/EN

Bumblebee simplifies building eBPF tools and allows you to package, distribute, and run them anywhere.Just focus on the eBPF portion of your code and BumbleBee automates away the boilerpart, including the userspace code.

Geeting the eBPF tool chain “just right” is hard. BumbleBee automates the build process and lets you to focus on the code. BumbleBee packages your eBPF code as an OCI image so you can distribute it across your infrastructure.

With BumbleBee, you focus on your eBPF code and run it anywhere. BumbleBee also builds the userspace code and can expose the eBPF maps as logs, metrics, and histograms. BumbleBee leverages BTP introspection to know what types to display.

* https://github.com/solo-io/bumblebee

BumbleBee helps to build, run and distribute eBPF programs using OCI images. It allows you to focus on writing eBPF code, while taking care of the user space components - automatically exposing your data as metrics or logs.

1. Prepare the environment
2. Create the program
   1. First thing to do is to choose the programming language.
   2. Now that we have selected the language to use, we will be prompted to select the type of program you want to create. As eBPF enables you to write programs that can hook into essentially any kernel functionality, there are several "types" of programs you can create.
   3. Next you will be asked for the type of global map you would like to use. Maps are the instrument through which eBPF user space, and kernel space programs are able to communicate with each other.
   4. After deciding on a map type, you will be asked to decide on an output format. Normally developing eBPF applications requires writing user space and kernel space code (with bee you only need to develop the kernel space code, and then bee can automatically handle and output the data from your eBPF maps).
3. Run the program

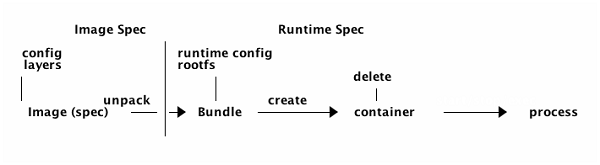
OCI SPECIFICATION -> Principio del CompileOnce-RunEverywhere

* https://alibaba-cloud.medium.com/open-container-initiative-oci-specifications-375b96658f55

OCI (Open Container Initiative) is an industry collaborated effort to define open containers specifications regarding container image format and runtime.

OCI has two specs, an Image spec and a Runtime spec.

The diagram illustrates what they cover and how they interact.



An OCI image will be downloaded from somewhere and then it will be unpacked into an OCI Runtime filesystem bundle. From that point, the OCI Runtime Bundle will be run by an OCI Runtime. The Runtime Specification defines how to run a “filesystem bundle”.

<https://www.youtube.com/watch?v=os2f0yfcgIU>

First do example

Then documentation