Libbpf Documentation. libbpf - BPF library

# **libbpf Overview[](https://libbpf.readthedocs.io/en/latest/libbpf_overview.html" \l "libbpf-overview" \o "Link to this heading)**

libbpf is a C-based library containing a BPF loader that takes compiled BPF object files and prepares and loads them into the Linux kernel. libbpf takes the heavy lifting of loading, verifying, and attaching BPF programs to various kernel hooks, allowing BPF application developers to focus only on BPF program correctness and performance.

The following are the high-level features supported by libbpf:

Provides high-level and low-level APIs for user space programs to interact with BPF programs. The low-level APIs wrap all the bpf system call functionality, which is useful when users need more fine-grained control over the interactions between user space and BPF programs.

Provides overall support for the BPF object skeleton generated by bpftool. The skeleton file simplifies the process for the user space programs to access global variables and work with BPF programs.

Provides BPF-side APIS, including BPF helper definitions, BPF maps support, and tracing helpers, allowing developers to simplify BPF code writing.

Supports BPF CO-RE mechanism, enabling BPF developers to write portable BPF programs that can be compiled once and run across different kernel versions.

This document will delve into the above concepts in detail, providing a deeper understanding of the capabilities and advantages of libbpf and how it can help you develop BPF applications efficiently.

## **BPF App Lifecycle and libbpf APIs[](https://libbpf.readthedocs.io/en/latest/libbpf_overview.html" \l "bpf-app-lifecycle-and-libbpf-apis" \o "Link to this heading)**

A BPF application consists of one or more BPF programs (either cooperating or completely independent), BPF maps, and global variables. The global variables are shared between all BPF programs, which allows them to cooperate on a common set of data. libbpf provides APIs that user space programs can use to manipulate the BPF programs by triggering different phases of a BPF application lifecycle.

The following section provides a brief overview of each phase in the BPF life cycle:

****Open phase****: In this phase, libbpf parses the BPF object file and discovers BPF maps, BPF programs, and global variables. After a BPF app is opened, user space apps can make additional adjustments (setting BPF program types, if necessary; pre-setting initial values for global variables, etc.) before all the entities are created and loaded.

****Load phase****: In the load phase, libbpf creates BPF maps, resolves various relocations, and verifies and loads BPF programs into the kernel. At this point, libbpf validates all the parts of a BPF application and loads the BPF program into the kernel, but no BPF program has yet been executed. After the load phase, it’s possible to set up the initial BPF map state without racing with the BPF program code execution.

****Attachment phase****: In this phase, libbpf attaches BPF programs to various BPF hook points (e.g., tracepoints, kprobes, cgroup hooks, network packet processing pipeline, etc.). During this phase, BPF programs perform useful work such as processing packets, or updating BPF maps and global variables that can be read from user space.

****Tear down phase****: In the tear down phase, libbpf detaches BPF programs and unloads them from the kernel. BPF maps are destroyed, and all the resources used by the BPF app are freed.

## **BPF Object Skeleton File[](https://libbpf.readthedocs.io/en/latest/libbpf_overview.html" \l "bpf-object-skeleton-file" \o "Link to this heading)**

BPF skeleton is an alternative interface to libbpf APIs for working with BPF objects. Skeleton code abstract away generic libbpf APIs to significantly simplify code for manipulating BPF programs from user space. Skeleton code includes a bytecode representation of the BPF object file, simplifying the process of distributing your BPF code. With BPF bytecode embedded, there are no extra files to deploy along with your application binary.

You can generate the skeleton header file (.skel.h) for a specific object file by passing the BPF object to the bpftool. The generated BPF skeleton provides the following custom functions that correspond to the BPF lifecycle, each of them prefixed with the specific object name:

<name>\_\_open() – creates and opens BPF application (<name> stands for the specific bpf object name)

<name>\_\_load() – instantiates, loads,and verifies BPF application parts

<name>\_\_attach() – attaches all auto-attachable BPF programs (it’s optional, you can have more control by using libbpf APIs directly)

<name>\_\_destroy() – detaches all BPF programs and frees up all used resources

Using the skeleton code is the recommended way to work with bpf programs. Keep in mind, BPF skeleton provides access to the underlying BPF object, so whatever was possible to do with generic libbpf APIs is still possible even when the BPF skeleton is used. It’s an additive convenience feature, with no syscalls, and no cumbersome code.

### **Other Advantages of Using Skeleton File[](https://libbpf.readthedocs.io/en/latest/libbpf_overview.html" \l "other-advantages-of-using-skeleton-file" \o "Link to this heading)**

BPF skeleton provides an interface for user space programs to work with BPF global variables. The skeleton code memory maps global variables as a struct into user space. The struct interface allows user space programs to initialize BPF programs before the BPF load phase and fetch and update data from user space afterward.

The skel.h file reflects the object file structure by listing out the available maps, programs, etc. BPF skeleton provides direct access to all the BPF maps and BPF programs as struct fields. This eliminates the need for string-based lookups with bpf\_object\_find\_map\_by\_name() and bpf\_object\_find\_program\_by\_name() APIs, reducing errors due to BPF source code and user-space code getting out of sync.

The embedded bytecode representation of the object file ensures that the skeleton and the BPF object file are always in sync.

## **BPF Helpers[](https://libbpf.readthedocs.io/en/latest/libbpf_overview.html" \l "bpf-helpers" \o "Link to this heading)**

libbpf provides BPF-side APIs that BPF programs can use to interact with the system. The BPF helpers definition allows developers to use them in BPF code as any other plain C function. For example, there are helper functions to print debugging messages, get the time since the system was booted, interact with BPF maps, manipulate network packets, etc.

For a complete description of what the helpers do, the arguments they take, and the return value, see the [bpf-helpers](https://man7.org/linux/man-pages/man7/bpf-helpers.7.html) man page.

## **BPF CO-RE (Compile Once – Run Everywhere)[](https://libbpf.readthedocs.io/en/latest/libbpf_overview.html" \l "bpf-co-re-compile-once-run-everywhere" \o "Link to this heading)**

BPF programs work in the kernel space and have access to kernel memory and data structures. One limitation that BPF applications come across is the lack of portability across different kernel versions and configurations. [BCC](https://github.com/iovisor/bcc/) is one of the solutions for BPF portability. However, it comes with runtime overhead and a large binary size from embedding the compiler with the application.

libbpf steps up the BPF program portability by supporting the BPF CO-RE concept. BPF CO-RE brings together BTF type information, libbpf, and the compiler to produce a single executable binary that you can run on multiple kernel versions and configurations.

To make BPF programs portable libbpf relies on the BTF type information of the running kernel. Kernel also exposes this self-describing authoritative BTF information through sysfs at /sys/kernel/btf/vmlinux.

You can generate the BTF information for the running kernel with the following command:

$ bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h

The command generates a vmlinux.h header file with all kernel types (BTF types) that the running kernel uses. Including vmlinux.h in your BPF program eliminates dependency on system-wide kernel headers.

libbpf enables portability of BPF programs by looking at the BPF program’s recorded BTF type and relocation information and matching them to BTF information (vmlinux) provided by the running kernel. libbpf then resolves and matches all the types and fields, and updates necessary offsets and other relocatable data to ensure that BPF program’s logic functions correctly for a specific kernel on the host. BPF CO-RE concept thus eliminates overhead associated with BPF development and allows developers to write portable BPF applications without modifications and runtime source code compilation on the target machine.

The following code snippet shows how to read the parent field of a kernel task\_struct using BPF CO-RE and libbf. The basic helper to read a field in a CO-RE relocatable manner is bpf\_core\_read(dst, sz, src), which will read sz bytes from the field referenced by src into the memory pointed to by dst.

*//...* **struct** **task\_struct** \*task = (void \*)bpf\_get\_current\_task(); **struct** **task\_struct** \*parent\_task; int err;

err = bpf\_core\_read(&parent\_task, **sizeof**(void \*), &task->parent); **if** (err) { */\* handle error \*/* }

*/\* parent\_task contains the value of task->parent pointer \*/*

In the code snippet, we first get a pointer to the current task\_struct using bpf\_get\_current\_task(). We then use bpf\_core\_read() to read the parent field of task struct into the parent\_task variable. bpf\_core\_read() is just like bpf\_probe\_read\_kernel() BPF helper, except it records information about the field that should be relocated on the target kernel. i.e, if the parent field gets shifted to a different offset within struct task\_struct due to some new field added in front of it, libbpf will automatically adjust the actual offset to the proper value.

## **Getting Started with libbpf[](https://libbpf.readthedocs.io/en/latest/libbpf_overview.html" \l "getting-started-with-libbpf" \o "Link to this heading)**

Check out the [libbpf-bootstrap](https://github.com/libbpf/libbpf-bootstrap) repository with simple examples of using libbpf to build various BPF applications.

See also [libbpf API documentation](https://libbpf.readthedocs.io/en/latest/api.html).

## **libbpf and Rust[](https://libbpf.readthedocs.io/en/latest/libbpf_overview.html" \l "libbpf-and-rust" \o "Link to this heading)**

If you are building BPF applications in Rust, it is recommended to use the [Libbpf-rs](https://github.com/libbpf/libbpf-rs) library instead of bindgen bindings directly to libbpf. Libbpf-rs wraps libbpf functionality in Rust-idiomatic interfaces and provides libbpf-cargo plugin to handle BPF code compilation and skeleton generation. Using Libbpf-rs will make building user space part of the BPF application easier. Note that the BPF program themselves must still be written in plain C.

## **libbpf logging[](https://libbpf.readthedocs.io/en/latest/libbpf_overview.html" \l "libbpf-logging" \o "Link to this heading)**

By default, libbpf logs informational and warning messages to stderr. The verbosity of these messages can be controlled by setting the environment variable LIBBPF\_LOG\_LEVEL to either warn, info, or debug. A custom log callback can be set using libbpf\_set\_print().