libbpf

unknown

Nov 22, 2023

CONTENTS

If you are looking to develop BPF applications using the libbpf library, this directory contains important documentation that you should read.

To get started, it is recommended to begin with the *libbpf Overview* document, which provides a high-level understanding of the libbpf APIs and their usage. This will give you a solid foundation to start exploring and utilizing the various features of libbpf to develop your BPF applications.

CHAPTER

ONE

LIBBPF OVERVIEW

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.

1.1 BPF App Lifecycle and libbpf APIs

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.

1.2 BPF Object Skeleton File

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.

1.2.1 Other Advantages of Using Skeleton File

- 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.

1.3 BPF Helpers

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 man page.

1.4 BPF CO-RE (Compile Once – Run Everywhere)

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 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.

1.5 Getting Started with libbpf

Check out the libbpf-bootstrap repository with simple examples of using libbpf to build various BPF applications.

See also libbpf API documentation.

1.6 libbpf and Rust

If you are building BPF applications in Rust, it is recommended to use the 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.

1.7 Additional Documentation

- Program types and ELF Sections
- API naming convention
- Building libbpf
- API documentation Convention

CHAPTER

TWO

PROGRAM TYPES AND ELF SECTIONS

The table below lists the program types, their attach types where relevant and the ELF section names supported by libbpf for them. The ELF section names follow these rules:

- `type` is an exact match, e.g. `SEC("socket")`
- `type+` means it can be either exact `SEC("type")` or well-formed `SEC("type/extras")` with a '/' separator between `type` and `extras`.

When `extras` are specified, they provide details of how to auto-attach the BPF program. The format of `extras` depends on the program type, e.g. `SEC("tracepoint/<category>/<name>")` for tracepoints or `SEC("usdt/<path>:<provider>:<name>")` for USDT probes. The extras are described in more detail in the footnotes.

Program Type	Attach Type	ELF Section Name	Sleepable
BPF_PROG_TYPE_CGROUP_	BPF_CGROUP_DEVICE	cgroup/dev	
BPF_PROG_TYPE_CGROUP_		cgroup/skb	
	BPF_CGROUP_INET_EGRES	cgroup_skb/egress	
	BPF_CGROUP_INET_INGRE	cgroup_skb/ingress	
BPF_PROG_TYPE_CGROUP_	BPF_CGROUP_GETSOCKOPT	cgroup/getsockopt	
	BPF_CGROUP_SETSOCKOPT	cgroup/setsockopt	
BPF_PROG_TYPE_CGROUP_	BPF_CGROUP_INET4_BIND	cgroup/bind4	
	BPF_CGROUP_INET4_CONN	cgroup/connect4	
	BPF_CGROUP_INET4_GETP	cgroup/getpeername4	
	BPF_CGROUP_INET4_GETS	cgroup/getsockname4	
	BPF_CGROUP_INET6_BIND	cgroup/bind6	
	BPF_CGROUP_INET6_CONN	cgroup/connect6	
	BPF_CGROUP_INET6_GETP	cgroup/getpeername6	
	BPF_CGROUP_INET6_GETS	cgroup/getsockname6	
	BPF_CGROUP_UDP4_RECVM	cgroup/recvmsg4	
	BPF_CGROUP_UDP4_SENDM	cgroup/sendmsg4	
	BPF_CGROUP_UDP6_RECVM	cgroup/recvmsg6	
	BPF_CGROUP_UDP6_SENDM	cgroup/sendmsg6	
	BPF_CGROUP_UNIX_CONNE	cgroup/connect_unix	
	BPF_CGROUP_UNIX_SENDM	cgroup/sendmsg_unix	
	BPF_CGROUP_UNIX_RECVM	cgroup/recvmsg_unix	
	BPF_CGROUP_UNIX_GETPE	cgroup/ getpeername_unix	
	BPF_CGROUP_UNIX_GETSO	cgroup/ getsockname_unix	
BPF_PROG_TYPE_CGROUP_	BPF_CGROUP_INET4_POST	cgroup/post_bind4	
	BPF_CGROUP_INET6_POST	cgroup/post_bind6	
	BPF_CGROUP_INET_SOCK_	cgroup/sock_create	

continues on next page

Table 1 – continued from previous page

Program Type	Attach Type	ELF Section Name	Sleepable
		cgroup/sock	
	BPF_CGROUP_INET_SOCK_	cgroup/sock_release	
BPF_PROG_TYPE_CGROUP_	BPF_CGROUP_SYSCTL	cgroup/sysctl	
BPF_PROG_TYPE_EXT		freplace+[1]	
BPF_PROG_TYPE_FLOW_DI	BPF_FLOW_DISSECTOR	flow_dissector	
BPF_PROG_TYPE_KPROBE		kprobe+[2]	
		kretprobe+[Page 9, 2]	
		ksyscall+[3]	
		kretsyscall+[Page 9, 3]	
		uprobe+[4]	
		uprobe.s+[Page 9, 4]	Yes
		uretprobe+[Page 9, 4]	
		uretprobe.s+[Page 9, 4]	Yes
		usdt+[5]	
	BPF_TRACE_KPROBE_MULT	kprobe.multi+[6]	
		kretprobe.multi+[Page 9, 6]	
BPF_PROG_TYPE_LIRC_MO	BPF_LIRC_MODE2	lirc_mode2	
BPF_PROG_TYPE_LSM	BPF_LSM_CGROUP	lsm_cgroup+	
	BPF_LSM_MAC	lsm+[7]	
		lsm.s+[Page 9, 7]	Yes
BPF_PROG_TYPE_LWT_IN		lwt_in	
BPF_PROG_TYPE_LWT_OUT		lwt_out	
BPF_PROG_TYPE_LWT_SEG		lwt_seg6local	
BPF_PROG_TYPE_LWT_XMI		lwt_xmit	
BPF_PROG_TYPE_PERF_EV		perf_event	
BPF_PROG_TYPE_RAW_TRACEPOINT_WRITABLE		raw_tp.w+[8]	
		raw_tracepoint.w+	
BPF_PROG_TYPE_RAW_TRACEPOINT		raw_tp+[Page 9, 8]	
		raw_tracepoint+	
BPF_PROG_TYPE_SCHED_A		action	
BPF_PROG_TYPE_SCHED_CLS		classifier	
		tc	
BPF_PROG_TYPE_SK_LOOK	BPF_SK_LOOKUP	sk_lookup	
BPF_PROG_TYPE_SK_MSG	BPF_SK_MSG_VERDICT	sk_msg	
BPF_PROG_TYPE_SK_REUS	BPF_SK_REUSEPORT_SELE	sk_reuseport/migrate	
	BPF_SK_REUSEPORT_SELE	sk_reuseport	
BPF_PROG_TYPE_SK_SKB		sk_skb	
	BPF_SK_SKB_STREAM_PAR	sk_skb/stream_parser	
	BPF_SK_SKB_STREAM_VER	sk_skb/stream_verdict	
BPF_PROG_TYPE_SOCKET_		socket	
BPF_PROG_TYPE_SOCK_OP	BPF_CGROUP_SOCK_OPS	sockops	
BPF_PROG_TYPE_STRUCT_		struct_ops+	
BPF_PROG_TYPE_SYSCALL		syscall	Yes
BPF_PROG_TYPE_TRACEPOINT		tp+[9]	
		tracepoint+[Page 9, 9]	
BPF_PROG_TYPE_TRACING	BPF_MODIFY_RETURN	fmod_ret+[Page 9, 1]	
		fmod_ret.s+[Page 9, 1]	Yes
	BPF_TRACE_FENTRY	fentry+[Page 9, 1]	

continues on next page

Table 1 – continued from previous page

Program Type	Attach Type	ELF Section Name	Sleepable
		fentry.s+[Page 9, 1]	Yes
	BPF_TRACE_FEXIT	fexit+[1]	
		fexit.s+[1]	Yes
	BPF_TRACE_ITER	iter+[10]	
		iter.s+[10]	Yes
	BPF_TRACE_RAW_TP	tp_btf+[1]	
BPF_PROG_TYPE_XDP	BPF_XDP_CPUMAP	xdp.frags/cpumap	
		xdp/cpumap	
	BPF_XDP_DEVMAP	xdp.frags/devmap	
		xdp/devmap	
	BPF_XDP	xdp.frags	
		xdp	

[1] The fentry attach format is fentry[.s]/<function>.

[2] The kprobe attach format is kprobe/<function>[+<offset>]. Valid characters for function are a-zA-Z0-9_. and offset must be a valid non-negative integer.

[3] The ksyscall attach format is ksyscall/<syscall>.

[4] The uprobe attach format is uprobe[.s]/<path>:<function>[+<offset>].

[5] The usdt attach format is usdt/<path>:<provider>:<name>.

[6] The kprobe.multi attach format is kprobe.multi/<pattern> where pattern supports * and ? wildcards. Valid characters for pattern are a-zA-Z0-9_.*?.

[7] The lsm attachment format is lsm[.s]/<hook>.

[8] The raw_tp attach format is raw_tracepoint[.w]/<tracepoint>.

[9] The tracepoint attach format is tracepoint/<category>/<name>.

[10] The iter attach format is iter[.s]/<struct-name>.

CHAPTER

THREE

API NAMING CONVENTION

libbpf API provides access to a few logically separated groups of functions and types. Every group has its own naming convention described here. It's recommended to follow these conventions whenever a new function or type is added to keep libbpf API clean and consistent.

All types and functions provided by libbpf API should have one of the following prefixes: `bpf_`, `btf_`, `libbpf_`, `btf_dump_`, `ring_buffer_`, `perf_buffer_`.

3.1 System call wrappers

System call wrappers are simple wrappers for commands supported by sys_bpf system call. These wrappers should go to `bpf.h` header file and map one to one to corresponding commands.

For example `bpf_map_lookup_elem` wraps `BPF_MAP_LOOKUP_ELEM` command of sys_bpf, `bpf_prog_attach` wraps `BPF_PROG_ATTACH`, etc.

3.2 Objects

Another class of types and functions provided by libbpf API is "objects" and functions to work with them. Objects are high-level abstractions such as BPF program or BPF map. They're represented by corresponding structures such as `struct bpf_object`, `struct bpf_program`, `struct bpf_map`, etc.

Structures are forward declared and access to their fields should be provided via corresponding getters and setters rather than directly.

These objects are associated with corresponding parts of ELF object that contains compiled BPF programs.

For example `struct bpf_object` represents ELF object itself created from an ELF file or from a buffer, `struct bpf_program` represents a program in ELF object and `struct bpf_map` is a map.

Functions that work with an object have names built from object name, double underscore and part that describes function purpose.

For example `bpf_object__open` consists of the name of corresponding object, `bpf_object`, double underscore and `open` that defines the purpose of the function to open ELF file and create `bpf_object` from it.

All objects and corresponding functions other than BTF related should go to `libbpf.h`. BTF types and functions should go to `btf.h`.

3.3 Auxiliary functions

Auxiliary functions and types that don't fit well in any of categories described above should have `libbpf_` prefix, e.g. `libbpf_get_error` or `libbpf_prog_type_by_name`.

3.4 ABI

libbpf can be both linked statically or used as DSO. To avoid possible conflicts with other libraries an application is linked with, all non-static libbpf symbols should have one of the prefixes mentioned in API documentation above. See API naming convention to choose the right name for a new symbol.

3.5 Symbol visibility

libbpf follow the model when all global symbols have visibility "hidden" by default and to make a symbol visible it has to be explicitly attributed with `LIBBPF_API` macro. For example:

```
LIBBPF_API int bpf_prog_get_fd_by_id(__u32 id);
```

This prevents from accidentally exporting a symbol, that is not supposed to be a part of ABI what, in turn, improves both libbpf developer- and user-experiences.

3.6 ABI versioning

To make future ABI extensions possible libbpf ABI is versioned. Versioning is implemented by `libbpf.map` version script that is passed to linker.

Version name is `LIBBPF_` prefix + three-component numeric version, starting from `0.0.1`.

Every time ABI is being changed, e.g. because a new symbol is added or semantic of existing symbol is changed, ABI version should be bumped. This bump in ABI version is at most once per kernel development cycle.

For example, if current state of `libbpf.map` is:

```
LIBBPF_0.0.1 {
        global:
                bpf_func_a;
                bpf_func_b;
        local:
                \*;
};
```

, and a new symbol `bpf_func_c` is being introduced, then `libbpf.map` should be changed like this:

```
LIBBPF_0.0.1 {
        global:
                bpf_func_a;
                bpf_func_b;
        local:
                \*;
};
```

(continues on next page)

(continued from previous page)

```
LIBBPF_0.0.2 {
        global:
                bpf_func_c;
} LIBBPF_0.0.1;
```

, where new version `LIBBPF_0.0.2` depends on the previous `LIBBPF_0.0.1`.

Format of version script and ways to handle ABI changes, including incompatible ones, described in details in [1].

3.7 Stand-alone build

Under https://github.com/libbpf/libbpf there is a (semi-)automated mirror of the mainline's version of libbpf for a stand-alone build.

However, all changes to libbpf's code base must be upstreamed through the mainline kernel tree.

CHAPTER

FOUR

API DOCUMENTATION CONVENTION

The libbpf API is documented via comments above definitions in header files. These comments can be rendered by doxygen and sphinx for well organized html output. This section describes the convention in which these comments should be formatted.

Here is an example from btf.h:

```
/**
 * @brief **btf__new()** creates a new instance of a BTF object from the raw
 * bytes of an ELF's BTF section
 * @param data raw bytes
 * @param size number of bytes passed in `data`
 * @return new BTF object instance which has to be eventually freed with
 * **btf__free()**
 *
 * On error, error-code-encoded-as-pointer is returned, not a NULL. To extract
 * error code from such a pointer `libbpf_get_error()` should be used. If
 * `libbpf_set_strict_mode(LIBBPF_STRICT_CLEAN_PTRS)` is enabled, NULL is
 * returned on error instead. In both cases thread-local `errno` variable is
 * always set to error code as well.
 */
```

The comment must start with a block comment of the form '/**'.

The documentation always starts with a @brief directive. This line is a short description about this API. It starts with the name of the API, denoted in bold like so: **api_name**. Please include an open and close parenthesis if this is a function. Follow with the short description of the API. A longer form description can be added below the last directive, at the bottom of the comment.

Parameters are denoted with the @param directive, there should be one for each parameter. If this is a function with a non-void return, use the @return directive to document it.

4.1 License

libbpf is dual-licensed under LGPL 2.1 and BSD 2-Clause.

4.2 Links

[1] **https://www.akkadia.org/drepper/dsohowto.pdf**
(Chapter 3. Maintaining APIs and ABIs).

CHAPTER

FIVE

BUILDING LIBBPF

libelf and zlib are internal dependencies of libbpf and thus are required to link against and must be installed on the system for applications to work. pkg-config is used by default to find libelf, and the program called can be overridden with PKG_CONFIG.

If using pkg-config at build time is not desired, it can be disabled by setting NO_PKG_CONFIG=1 when calling make.

To build both static libbpf.a and shared libbpf.so:

```
$ cd src
$ make
```

To build only static libbpf.a library in directory build/ and install them together with libbpf headers in a staging directory root/:

```
$ cd src
$ mkdir build root
$ BUILD_STATIC_ONLY=y OBJDIR=build DESTDIR=root make install
```

To build both static libbpf.a and shared libbpf.so against a custom libelf dependency installed in /build/root/ and install them together with libbpf headers in a build directory /build/root/:

```
$ cd src
$ PKG_CONFIG_PATH=/build/root/lib64/pkgconfig DESTDIR=/build/root make
```

All general BPF questions, including kernel functionality, libbpf APIs and their application, should be sent to bpf@vger.kernel.org mailing list. You can subscribe to the mailing list search its archive. Please search the archive before asking new questions. It may be that this was already addressed or answered before.