Architecture of the library

`ELF -> Specifications -> Objects -> Links`

ELF

BPF is usually produced by using Clang to compile a subset of C. Clang outputs an ELF file which contains program byte code (aka BPF), but also metadata for maps used by the program. The metadata follows the conventions set by libbpf shipped with the kernel. Certain ELF sections have special meaning and contain structures defined by libbpf. Newer versions of clang emit additional metadata in BPF Type Format (aka BTF).

The library aims to be compatible with libbpf so that moving from a C toolchain to a Go one creates little friction. To that end, the ELF reader is tested against the Linux selftests and avoids introducing custom behaviour if possible.

The output of the ELF reader is a `CollectionSpec` which encodes all of the information contained in the ELF in a form that is easy to work with in Go.

BTF

The BPF Type Format describes more than just the types used by a BPF program. It includes debug aids like which source line corresponds to which instructions and what global variables are used.

BTF parsing lives in a separate internal package since exposing it would mean an additional maintenance burden, and because the API still has sharp corners. The most important concept is the `btf.Type` interface, which also describes things that aren't really types like `.rodata` or `.bss` sections. `btf.Type`s can form cyclical graphs, which can easily lead to infinite loops if one is not careful. Hopefully a safe pattern to work with `btf.Type` emerges as we write more code that deals with it.

Specifications

`CollectionSpec`, `ProgramSpec` and `MapSpec` are blueprints for in-kernel objects and contain everything necessary to execute the relevant `bpf(2)` syscalls. Since the ELF reader outputs a `CollectionSpec` it's possible to modify clang-compiled BPF code, for example to rewrite constants. At the same time the asm package provides an assembler that can be used to generate `ProgramSpec` on the fly.

Creating a spec should never require any privileges or be restricted in any way, for example by only allowing programs in native endianness. This ensures that the library stays flexible.

Objects

`Program` and `Map` are the result of loading specs into the kernel. Sometimes loading a spec will fail because the kernel is too old, or a feature is not enabled. There are multiple ways the library deals with that:

- Fallback: older kernels don't allow naming programs and maps. The library automatically detects support for names, and omits them during load if necessary. This works since name is primarily a debug aid.
- Sentinel error: sometimes it's possible to detect that a feature isn't available. In that case the library will return an error wrapping `ErrNotSupported`. This is also useful to skip tests that can't run on the current kernel.

Once program and map objects are loaded they expose the kernel's low-level API, e.g. `NextKey`. Often this API is awkward to use in Go, so there are safer wrappers on top of the low-level API, like `MapIterator`. The low-level API is useful when our higher-level API doesn't support a particular use case.

Links

BPF can be attached to many different points in the kernel and newer BPF hooks tend to use bpf_link to do so. Older hooks unfortunately use a combination of syscalls, netlink messages, etc. Adding support for a new link type should not pull in large dependencies like netlink, so XDP programs or tracepoints are out of scope.