Making Libbpf and Bpftool Cross-Platform

Dave Thaler

eBPF Runtime Platforms

Platform →	Linux		MacOSX		Android		FreeBSD		Windows		TockOS (embedded)	
↓ Project	Kernel	User	Kernel	User	Kernel	User	Kernel	User	Kernel	User	Kernel	User
Linux	2014											
uBPF		2015										
rbpf		2017		2017						2018		
Android					2017							
Generic eBPF	2017	2017		2017			2017	2017				
eBPF for Windows									2021			
Tock											2021	

Goal: Cross-plat libbpf and bpftool

- Why:
  - Get more people making them better, rather than reimplementing them
  - Allow more easily writing cross-platform apps that use libbpf
- How:
  - Refactor to split platform-agnostic functionality from platform-specific stuff

Categories of issues

1. Compiler-specific code
2. Platform-specific code
3. ebpf feature-specific code
4. Prog type / hook specific code
5. Repositories and CI/CD

Issue 1: Compiler-specific code (1/2)

- Compiler-specific pragmas and attributes
  - #pragma **GCC** poison reallocarray
  - **__attribute__**((alias("bpf_prog_attach_opts")))
- Compiler specific language extensions
  - "= {}" is not legal in the C standard
  - "#define LIST_POISON1 **((void *)** 0x100 **+** POISON_POINTER_DELTA)" is not legal in the C standard
  - #define NEXT_ARG() **({** argc--; argv++; if (argc < 0) usage(); **})**
- Assumptions about type sizes
  - **size_t** new_cap = **1UL** << new_cap_bits;

Proposal:

1. Adhere to standard C whenever possible, or at least features supported by all relevant compilers
2. Avoid unnecessary assumptions about type sizes
3. Cross-plat files should not hard code any compiler-specific pragmas or attributes
4. Move compiler specific defines to "compiler.h", for use by cross-plat files
5. Each compiler will have its own "compiler.h" in a separate subdirectory
   - e.g., gcc/compiler.h (but currently have linux/compiler.h in github.com/libbpf/libbpf), msvc/compiler.h
6. Use include path list to specify compiler directory, not hard coded in #include
   - #include "compiler.h"
7. Avoid compiler specific ifdefs

Issue 1: Example (2/2)

- gnuc/compiler.h:

```
#define LIBBPF_DEPRECATED(msg) __attribute__((deprecated(msg)))
#define LIBBPF_ALIAS(a) __attribute__((alias(a)))
…
```

Issue 2: Platform-specific code (1/2)

- Direct inclusion of platform specific headers
  - #include <linux/limits.h>
- Non-ebpf-specific platform features that vary by platform
  - rlimit, netlink, etc.
- Function implementations that vary by platform (see next slide)

- Proposal:
  1. Put platform specific includes/defines in a header file like “platform.h”, for use by cross-plat files
  2. Each platform will have its own “platform.h” in a separate subdirectory
     - e.g., linux/platform.h, windows/platform.h
  3. Use include path list to specify platform directory, not hard coded in #include
     - #include “platform.h”
  4. Avoid platform specific ifdefs
  5. Do same for functions whose implementation varies by platform in .c files

Background regarding syscalls (2/2)

- Q (Lorenz): Have you decided what your ABI / API boundary is going to be? Is it raw syscalls or the libbpf C API?
- A: libbpf. bpf() is a shim over other libbpf APIs.
- Rationale:
  - syscall() does not exist on Windows, and ioctls work differently on Windows
  - FD's only exist in the userland C runtime, kernel uses HANDLE which is ptr size
  - bpf_attr used with bpf() contains FD's so isn't large enough to pass to kernel
  - Implementation of many libbpf APIs thus must contain a userspace step
- Takeaway is that code that directly uses syscalls is platform-specific

Issue 3: ebpf feature-specific code (1/2)

- New runtimes don't support all core ebpf features, at least at the same time
  - BTF, BPF fs, etc.
- eBPF features may thus vary by platform over time
- Some bpftool commands or args are feature specific
  - bpftool btf …
  - bpftool … --bpffs
- Same for libbpf:
  - LIBBPF_API struct btf *bpf_object__btf(const struct bpf_object *obj);

- When a libbpf API is not supported on a platform, should it
  a. Be absent? (my preference, based on things like IDE auto-completion)
  b. Be present and always return failure?
- When a bpftool command/arg is not supported on a platform, should it
  a. Be absent from help? (my preference)
  b. Be present but always show an error message when tried?

Issue 3: strawman proposal (2/2)

- Should feature-specific functions be
  a. In a separate file from other functions
  b. Surrounded in a feature specific ifdef like HAVE_BTF_SUPPORT
- What about code that enumerates cmds/args incl. feature specific ones, e.g. help text?
  - Example: bpftool main cmds[] and “bpftool help” output

- Strawman:
  - Put feature-specific code in a separate file per feature as much as possible
  - In some cases, the separate file could even be in another repo (see later slide)
  - Platform-specific file could expose a (possibly no-op) function that walks through the set of features
  - After doing/showing anything that is platform-agnostic, enumeration calls a function implemented by each platform to pick up platform-specific variations

Issue 4: Prog type / hook specific code (1/2)

- The set of program types and attach types can vary by platform (and version)
  - LIBBPF_API int bpf_tc_hook_create(struct bpf_tc_hook *hook);
  - bpftool net help
    - Note: Only xdp and tc attachments are supported now.
    - For progs attached to cgroups, use "bpftool cgroup"
    - to dump program attachments. For program types
    - sk_{filter,skb,msg,reuseport} and lwt/seg6, please
    - consult iproute2.
- Bpftool and libbpf today also hard code list of prog types, attach types, etc.
- In eBPF-for-Windows (and possibly other runtimes), the list is not fixed at compile time
  - Some additions don’t require libbpf/bpftool “code” changes, e.g., additional attach type

- Proposal:
  - Create platform-specific function to retrieve list
  - Implement in platform-specific file
  - For any prog type specific cmds/args, handle like ebpf features as discussed previously

Integer values (prog types, etc.) (2/2)

- eBPF for Windows allows ebpf hooks & helpers to be implemented in additional drivers that are installed/loaded post boot
  - Runtime-introspection is used to register them with verifier and execution context
  - Integer values are of course centrally coordinated, but prog types & attach types also have UUIDs that can be used during development before getting an integer assigned
    - libbpf_prog_type_by_name, libbpf_attach_type_by_name
- Currently integer values may vary by platform
  - They typically don’t appear literally in source code for ebpf progs or apps
  - No need to coordinate across all platforms to add a platform-specific one
- Libbpf has APIs to convert name to prog type, but bpftool hard codes the reverse itself (prog_type_name[]), which should probably be a libbpf API

Issue 5: Repositories (1/2)

- Libbpf and bpftool have their own repos now but are just mirrors of bpf-next Linux source tree
- Alexei: “All patches have to go via bpf@vger and land via bpf/bpf-next trees. Non-linux patches would be awesome to see.”
- Issue: using bpf@vget and bpf/bpf-next is a huge hurdle for other runtimes used to github
  - Not just due to risk of “some noise from linux and GPL fanatics”
  - Hurdles provide incentive to fork or reimplement, which is not good for ebpf as a whole
- Do we really WANT lots of non-linux files in the linux source tree?
  - They may require other SDKs or repos (e.g., ebpf-for-windows) as prerequisites to build (see next slide)

- Strawman proposal:
  - Put files for other runtimes in separate repos (could even be the repo for that runtime)
  - Linux platform files stay in Linux source tree as is
  - For now, keep platform agnostic files in the Linux source tree though this may or may not make sense longer term
  - Use existing mirrors to add additional github workflows to as needed

Issue 5: CI/CD testing (2/2)

- Daniel: “We also have BPF selftests which run on every submitted patch to the kernel and a lot of them involve libbpf as well [0]. My other hope is that if Windows relies on the very same libbpf, perhaps this would also allow for portability/mock testing the available Windows hooks on Linux and vice versa”
- When making a change in platform-specific code, when does CI/CD build/test happen on each runtime?
  - Propose that platform-specific code stays in platform-specific repo with its own CI/CD build/testing
  - Linux does not build/test Windows-specific code or vice versa
- When making a change in platform-agnostic code, when does CI/CD build/test happen on each runtime?
  - A) Each runtime is tested (e.g., via github workflow) before a core change is merged
    - Harder to coordinate if in Linux repo, and “libbpf has a higher rate of changes than the kernel” (Alexei)
  - B) Linux runtime is tested before merge, other runtimes are tested after merge
    - Higher risk of regression for other runtimes and creates incentive to fork/re-implement