On getting tc classifier fully programmable with cls_bpf.

Daniel Borkmann
Cisco
Switzerland
daniel@iogearbox.net

Abstract

Berkely Packet Filter, short BPF, is an instruction set architecture that was designed long ago in 1993 [18] [1] as a generic packet filtering solution for applications such as `libpcap`/`tcpdump` and is long present in Linux kernels, where it is also being used outside of networking, e.g. in seccomp BPF [15] for system call filtering.

In recent years, the Linux community replaced the nowadays referred to as classic BPF (cBPF) interpreter inside the kernel with a new instruction set architecture called ”extended BPF” (eBPF) [21] [23] [22] [24] that brings far more flexibility and programmability aspects compared to its predecessor, and new use cases along with it such as tracing [27] or more recently KCM [17]. Along with the interpreter replacement, also the just-in-time (JIT) compiler has been upgraded to translate eBPF [25] for running programs with native performance.

With eBPF support that has been added to the kernel's `cls_bpf` classifier [8] in the traffic control layer [8], `tc` has gained a powerful member to program Linux' data plane with a tight integration to the kernel's networking stack and related tooling as well as different underlying programming paradigms compared to its cBPF predecessor.

In this paper, we provide a basic overview of eBPF, its interaction with `tc`, and discuss some of the recent work that went into eBPF that has been done by the Linux networking community. The intention of this paper is not to provide complete coverage of all eBPF aspects, but rather tries to be a informational starting point for people interested in its architecture and relation with `tc`.

Keywords

eBPF, cls_bpf, tc, programmable datapath, Linux kernel

Introduction

The classic BPF or cBPF architecture has been part of the Linux kernel for many years. Its most prominent user has been `PF_PACKET` sockets, where cBPF is used as a generic, fast and safe solution to implement packet parsing at an early point in the `PF_PACKET` receive path. One of the primary goals in relation to a safe execution was to not by any means make the kernel's operation unstable from injected untrusted user programs.

cBPF's architecture [18] is 32 bit and has two primary registers available `A` and `X`, a 16 words scratch space usually referred to as `M`, and implicitly a program counter. Its design was heavily driven by the packet parsing use case. Register `A` is the main register, also referred to as accumulator, where most operations are performed such as alu, load, stores or comparison operations for jumps. `X` was mostly used as a temporary register, and is also used for relative loads of packet contents. With cBPF, packet contents can only be read, but not modified.

cBPF has eight different instruction classes, namely `ld`, `ldx`, `st`, `stx`, `alu`, `jmp`, `ret` and `misc`. First ones denote load and store instructions involving `A` or `X`, respectively. Next to alu and jump instructions, there are return instructions for terminating cBPF execution, and few miscellaneous instruction to transfer contents of `A` and `X`.

cBPF supports only forward jumps, a maximum of 4096 instructions per cBPF program and the code is statically verified inside the kernel before execution. Overall, `bpf_asm` tool [5] counts 33 instructions, 11 addressing modes, and 16 Linux specific cBPF extensions.

The semantics of the cBPF program are defined by the subsystem making use of it. Today, cBPF found many use cases beyond `PF_PACKET` sockets due to its generic, minimal nature and fast execution. seccomp BPF [15] parts were added in 2012 for the purpose of having a safe and fast way of system call filtering. In the networking domain, cBPF can be used as socket filters for most protocols such as TCP, UDP, netlink, etc, as a fanout demuxing facility [14] [13] for `PF_PACKET` sockets, for socket demuxing with `SO_REUSEPORT` [16], as load balancing in team driver [19], for the here discussed `tc` subsystem as a classifier [6] and action [20], and a couple of other miscellaneous users.

Programmability aspects changed radically when eBPF has successively been introduced into the kernel since its first merge and later follow up work since 2014.

eBPF Architecture

Like cBPF, eBPF can be regarded as a minimalistic ”virtual” machine construct [21]. The machine it abstracts has only a few registers, stack space, an implicit program counter and a helper function concept that allows for side effects with the rest of the kernel. It operates event driven on inputs (”context”), which in case of `tc` is a `skb`, for example, traversing ingress or egress direction the traffic control layer of a given device.

eBPF has 11 registers (`R0` to `R10`) that are 64 bit wide with 32 bit subregisters. The instruction set has a fixed instruction size of 64 bit, and can be regarded as a mixture of `cBPF`, `x86_64`, `arm64` and `risc`, designed to more closely resemble underlying machine opcodes in order to ease eBPF JIT compilation.

Some of the cBPF remains have been carried over into eBPF to make in-kernel cBPF to eBPF migrations easier. Generally, eBPF has a stable ABI for user space, similarly as in cBPF case.

Internally, the Linux kernel ships with an eBPF interpreter and JIT compiler for (currently) `x86_64`, `s390`, `arm64`. There are a couple of other architectures such as `ppc`, `sparc`, `arm` and `mips` that still have not converted their cBPF JIT to an eBPF JIT yet, thus loading native eBPF code is required to be run through the interpreter in these cases. However, loading cBPF code through the available kernel interfaces will, if it cannot be handled by the remaining cBPF JITs for these architectures, be in-kernel migrated into eBPF instructions and either JIT compiled through an eBPF JIT back end or be handled by the interpreter. One such user, for example, was seccomp BPF, that immediately had a benefit of the eBPF migration as it could be JIT compiled as a side effect of the eBPF introduction.

The eBPF instruction encoding consists of 8 bit `code`, 8 bit `dst_reg`, 8 bit `src_reg`, signed 16 bit `offset` and, last but not least, signed 32 bit `imm` member. `code` holds the actual instruction code, `dst_reg` and `src_reg` denote register numbers (`R0` to `R10`) to be used by the instruction, `offset` depending on the instruction class can either be a jump offset in case the related condition is evaluated as true, it can be a relative stack buffer offset for load/stores of registers into the stack, or in case of an `xadd` alu instruction, it can be an increment offset. `imm` carries the immediate value.

eBPF comes with a couple of new instructions, such as alu operations working on 64 bit, a signed shift operation, load/store of double words, a generic move operation for registers and immediate values, operators for endianness conversion, a call operation for invoking helper functions, and an atomic add (`xadd`) instruction.

Similarly as with cBPF, a maximum of 4096 instructions can be loaded into the kernel per program, and the passed instruction sequence is being statically verified in the kernel, which is necessary for rejecting programs that could otherwise destabilize the kernel, for example, through constructs such as infinite loops, pointer or data leaks, invalid memory accesses, etc. While cBPF allows only for forward jumps, eBPF allows for forward and limited backward jumps as far as a backward jump doesn't generate a loop, and thus guarantees that the program comes to halt.

eBPF has a couple of additional architectural concepts like helper functions, maps, tail calls, object pinning. In the next paragraphs, we are going to discuss each entity.

Helper Functions

Helper functions are a concept that lets eBPF programs consult a core kernel defined set of function calls in order to retrieve/push data from/to the kernel. Available helper functions may differ for each eBPF program type, for example, eBPF programs attached to sockets are only allowed to call into a subset of helpers as opposed to eBPF programs attached to the traffic control layer. Encapsulation and decapsulation helpers for flow-based tunneling constitute an example of functions that are only available to lower `tc` layers on ingress and egress.

Each helper function is implemented with a commonly shared function signature similar to system calls and is defined as `u64 foo(u64 r1, u64 r2, u64 r3, u64 r4, u64 r5)` with a fixed calling convention that `R0` contains the return value, `R1` to `R5` function arguments, `R6` to `R9` registers are callee saved, and `R10` acts as a read-only frame pointer used for stack space load/stores. There are a couple of advantages with this approach: while cBPF overloaded the load instruction in order to fetch data at an impossible packet offset to invoke auxiliary helper functions, each cBPF JIT needed to implement support for such a cBPF extension. In case of eBPF, each newly added helper function will be JIT compiled in a transparent and efficient way, meaning that the JIT compiler only needs to emit a `call` instruction since the register mapping is made in such a way that eBPF register assignments already match the underlying architecture's calling convention.

Mentioned function signature also allows the verifier to perform type checks. `struct bpf_func_proto` is used to hand all the necessary information that is needed to know about the helper to the verifier, so the verifier can make sure that expected types from the helper match with the current contents of the eBPF program's utilized registers. Argument types can range from passing in any kind of value up to restricted contents such as a pointer/size pair for the eBPF's stack buffer, which the helper should read from or write to. In the latter case, the verifier can also perform additional checks, for example, whether the buffer was initialized previously.

Maps

eBPF maps are another flexible entity which constitutes a part of the eBPF architecture. Maps are an efficient key/value store that reside in kernel space and are accessed through file descriptors from user space. Maps can be shared between multiple eBPF programs, but also between an eBPF program and user space. There are no limitations with regards to sharing, for example, a map could be shared among a `tc` related program and a tracing related one. Map back ends are provided by the core kernel and are of generic or specialized type (some specialized maps such as [28] may be only used for a given subsystem, though). Generic maps currently are available in form of an array or a hash table, both in per-CPU but also non-per-CPU flavours. Access to maps is realized from an eBPF program through previously described helper functions. From user space, maps can be managed through the `bpf(2)` system call. Creation of maps can only be done from user space via `bpf(2)`. This means that in case an eBPF program needs to populate one of the registers with a reference to the map in order to perform a map helper function call, it needs to encode the file descriptor value into the instruction, for example, helper macro `BPF_LD_MAP_FD(BPF_REG_1, fd)` would be such a case. The kernel recognizes this special `src_reg` case, can lookup the file descriptor from the file descriptor table to retrieve the actual eBPF map, and rewrite the instruction internally.

Object Pinning

eBPF maps and programs act as a kernel resource and can only be accessed through file descriptors, backed by anonymous inodes in the kernel. Advantages, but also a number of disadvantages come along with them: user space applications can make use of most file descriptor related APIs, file descriptor passing for Unix domain sockets work transparently, etc, but at the same time, file descriptors are limited to a processes' lifetime, which makes things like map sharing rather cumbersome. Thus it brings a number of complications for certain use cases such as `tc`, where `tc` sets up and loads the program into the kernel and terminates itself eventually. With that, also access to maps are unavailable from user space side, where it would otherwise have been useful, for example, when third party applications may wish to monitor or update map contents during eBPF program runtime. There were a couple of ideas on how to keep such file descriptors alive, one being to reuse `fuse` that would act as a proxy for `tc`. Thus, file descriptors are owned by the `fuse` implementation and tools like `tc` can fetch related file descriptors through Unix domain sockets.

However, this as well brings a number of issues. Deployments then additionally depend on `fuse` to be installed and need to run an additional daemon. Large scale deployments that try to maintain a minimalistic user space for saving resources might be unwilling to accept such additional dependencies. To resolve this, a minimal kernel space file system has been implemented [4] where eBPF map and programs can be *pinned*, a process we call *object pinning*. The `bpf(2)` system call has been extended with two new commands that can pin or retrieve a previously pinned object. For instance, tools like `tc` make use of this new facility [9] for sharing maps on ingress and egress. The eBPF-related file system keeps an instance per mount namespace, supports bind mounts, hard links, etc. It integrates seamlessly when spawning a new network namespace through `ip-netns`, and depending on use cases, different shared subtree semantics can be utilized.

Tail Calls

Another concept that can be used with eBPF is called tail calls [26]. Tail calls can be seen as a mechanism that allows one eBPF program to call another, without returning back to the old program. Such a call has minimal overhead as unlike function calls, it is implemented as a long jump, reusing the same stack frame. Such programs are verified independently of each other, thus for transferring state, either per-CPU maps as scratch buffers or `skb` fields such as `cb` area must be used. Only programs of the same type can be tail called, and they also need to match in terms of JIT compilation, thus either JIT compiled or only interpreted programs can be invoked, but not mixed together.

There are two components involved for realizing tail calls: the first part needs to setup a specialized map called *program array* that can be populated by user space with key/values where values are the file descriptors of the tail called eBPF programs, the second part is a `bpf_tail_call()` helper where the context, a reference to the program array and the lookup key is passed to. The kernel translates this helper call directly into a specialized eBPF instruction. Such a program array is write-only from user space side.

The kernel looks up the related eBPF program from the passed file descriptor and atomically replaces program pointers at the map slot. When no map entry has been found at the provided key, the kernel will just ”fall through” and continue execution of the old program with the instructions following the `bpf_tail_call()`. Tail calls are a powerful utility, for example, parsing network headers could be structured through tail calls. During runtime, functionality can be added or replaced atomically, and thus altering execution behaviour.

Security

eBPF has a couple of mitigation techniques to prevent intentional or unintentional corruption of program images through kernel bugs, that do not necessarily need to be BPF related. For architectures supporting `CONFG_DEBUG_SET_MODULE_RONX`, the kernel will lock eBPF interpreter images as read-only [2]. When JIT compiled, the kernel also locks the generated executable images as read-only and randomizes their start address to make guessing harder. The gaps in the images are filled with trap instructions (for example, on `x86_64` it is filled with `int3` opcodes) for catching such jump probes. eBPF will soon also have a constant blinding facility for unprivileged programs. For unprivileged programs, the verifier also imposes restrictions on helper functions that can be used, restrictions on pointers, etc, to make sure that no data leakage can occur.

LLVM

One important aspect of eBPF is how programs can be written. While cBPF has only a few options available such as the `libpcap`'s cBPF compiler, `bpf_asm`, or other means of hand crafting such programs, eBPF eases usage significantly by the possibility of implementing eBPF programs from higher level languages such as C (P4 language front ends exist as well).

`llvm` has an eBPF back end for emitting ELF files that contain eBPF instructions, front ends like `clang` can be used to craft programs. Compiling eBPF with `clang` is fairly easy by invoking `clang -O2 -target bpf -o bpf_prog.o -c bpf_prog.c`. Often quite useful is also `clang`'s option to output assembly through `clang -O2 -target bpf -o - -S -c bpf_prog.c` or tools like `readelf` to dump and analyze ELF sections and relocations. A typical workflow is to implement eBPF programs in C, to compile them with `clang` and pass to eBPF loaders such as `tc` that interact with classifiers like `cls_bpf`.

cls_bpf and eBPF

`cls_bpf` started out in 2013 [6] as a cBPF-based classifier, programmable through `bpf_asm`, `libpcap/tcpdump` or any other cBPF bytecode generator. The bytcode is then passed through the tc front end, which mainly sets up the netlink message to push the code down into cls_bpf.

Some time ago act_bpf [20] followed, which as any action in tc, can be attached to classifiers. act_bpf supporting cBPF bytecode as well has effectively the same functionality as cls_bpf with the only difference of their opcodes they return. While cls_bpf can return any tc classid (major/minor), act_bpf in contrast, returns a tc action opcode.

A downside at the introduction of act_bpf however was, that while being tied to cBPF only, the skb could not be mangled and would thus require further processing in the action pipeline by invoking tc actions such as act_pedit at the cost of additional performance penalty per packet.

eBPF support for cls_bpf and act_bpf got added shortly later with eBPF program types BPF_PROG_TYPE_SCHED_CLS [8] and BPF_PROG_TYPE_SCHED_ACT [7], respectively. The fast path for both types run under RCU and their main task is nothing more than just to invoke BPF_PROG_RUN(), which resolves to (*filter->bpf_func)(ctx, filter->insnsi), where the skb is to be processed as the BPF input context with either the eBPF interpreter (__bpf_prog_run()) selected as a function dispatch or the generated JIT image by one of the architecture provided JIT compilers.

Functions like cls_bpf_classify() are unaware of the underlying BPF flavour, so skbs pass through the same path for cBPF as well as eBPF classifier. One of the advantages of cls_bpf over various other tc classifiers is that it allows for efficient, non-linear classification (and integrated actions), meaning the BPF program can be tailored, so that a single parsing pass is enough to process skbs of different types. Historically, tc allows for multiple classifiers to be attached, and when a lack of match occurs, the next classifier in the chain is executed. It becomes inefficient when parts of the packet have to be re-parsed in the next classifiers over and over again. With cls_bpf this can be avoided easily with a single eBPF program, or in a eBPF tail call program construct, which allows for atomic replacements of parts of the packet parser. There, the program, based on the classification (or action) outcome, can return different classids or opcodes.

Working Modes

cls_bpf has two working modes for dealing with actions. Originally, tcf_exts_exec() has been invoked after classification, but as eBPF is more powerful than just doing classification, i.e. it can mangle packet contents, update checksums, etc by itself already, it has been decided to add a direct action (da) mode [3], which is a recommended fast path when working with cls_bpf.

In this mode, cls_bpf performs actions on the skb, and just returns a tc opcode, which eventually allows for having a compact, lightweight image for efficient skb processing, without needing to traverse multiple layers of indirection and list handling when using the full tc action engine. For eBPF, the classid can be stored in skb->tc_classid, and the action opcode is being returned. The latter also works well for simple cases with cBPF like drop actions.

Nevertheless, cls_bpf still allows for the administrator to use multiple classifiers in mixed modes (da and non-da) if the use case makes it necessary. We recommend, however, to make the fast path as compact as possible, thus for high performance workloads, a single tc eBPF cls_bpf classifier in da mode should be sufficient in the vast majority of cases.

Features

With eBPF there are number of possibilities to use in cls_bpf with regards to the context itself and provided helper functions. They are building blocks that can be tailored together for a given, specific use case.

For the context (skb here is of type struct __sk_buff), cls_bpf allows reads *and* writes to skb->mark, skb->priority, skb->tc_index, skb->cb[5], skb->tc_classid members, it allows reads to skb->len, skb->pkt_type, skb->queue_mapping, skb->protocol, skb->vlan_tci, skb->vlan_proto, skb->vlan_present, skb->ifindex (translates to netdev's ifindex) and skb->hash.

There are a number of helper functions available to be used from cls_bpf program types. These include among others, eBPF map access (lookup, update, deletion), invoking of eBPF tail calls, storing and loading (multi-)bytes into the skb for parsing and packet mangling, L3 and L4 checksum fix up helpers, encapsulation and decapsulation support. The latter provides helpers for vlan, but also supports the tunnel key infrastructure, where eBPF acts as a front end to feed tunnel related meta data on transmission for back end implementations like vxlan, geneve or gre. Similarly, the meta data retrieved on receive is stored in the tunnel key and can be read out from eBPF again. Such vxlan or geneve flow-based tunneling back end devices operate in *collect metadata* mode for this purpose.

skbs can also be redirected from cls_bpf either as egress through dev_queue_xmit(), or back into the ingress path from another device via dev_forward_skb(). There are currently two possibilities for redirection, either as a cloned skb during eBPF program runtime, or as a faster variant, where the skb doesn't need to be cloned. The latter requires cls_bpf to be run in da mode, where the return code is TC_ACT_REDIRECT that is supported by qdiscs like sch_clsact [12] on ingress and egress path. Here, the program fills a per-CPU scratch buffer with necessary redirection information during eBPF program runtime, and on return with the related opcode, the kernel will take care of the redirection through skb_do_redirect(). This facility acts as a performance optimization, so that the skb can be efficiently forwarded.

For debugging purposes, there is a bpf_trace_printk() helper also available for cls_bpf that allows dumping printk()-like messages into the trace pipe, which can be read through commands like tc exec bpf dbg. This turns out to be a useful facility for writing and debugging eBPF programs despite its limitations as a helper function: from five arguments that can be passed, the first two are inevitably format string related. Furthermore, clang needs to emit code that copies the given format string into the eBPF stack buffer first.

There are a couple of other helper functions available, for example, to read out the skb's cgroup classid (net_cls cgroup), to read the dst's routing realm (dst->tclassid), to fetch a random number (f.e. for sampling purposes), to retrieve the current CPU the skb is processed on, or to read out the time in nanoseconds (ktime_t).

cls_bpf can be attached to a couple of invocation points related to the traffic control subsystem. They can be categorized into three different hook types: the ingress hook, the recently introduced egress hook and the classification hook inside classful qdiscs on egress. The first two can be configured through sch_clsact [12] qdisc (or sch_ingress for the ingress-only part) and are invoked lockless under RCU context. The egress hook runs centrally from __dev_queue_xmit() before fetching the transmit queue from the device.

Front End

The tc cls_bpf iproute2 front end [10] [11] [9] does quite a bit of work in the background before pushing necessary data for cls_bpf into the kernel over netlink. It contains a common ELF loader back end for f_bpf (classifier), m_bpf (action), and e_bpf (exec) parts, so that commonly used code can be shared. When compiling an eBPF program with clang, it generates an object file in ELF format that is passed to tc for loading into the kernel. It is effectively a container for all the necessary data that tc needs to extract, relocate and load for getting the eBPF program hooked up in cls_bpf.

On startup, tc checks and if necessary mounts the bpf fs for object pinning by default to /sys/fs/bpf, loads and generates a hash table for pinning configuration in case maps are to be shared other than on a per-object or global scope.

After that, tc walks ELF sections of the object file. There are a couple of reserved section names, namely maps for eBPF map specifications (e.g. map type, key and value size, maximum elements, pinning, etc) and license for the licence string, specified similarly as in Linux kernel modules. Default entry section names are classifier and action for cls_bpf and act_bpf, respectively. tc fetches all ancillary sections first, including the ELF's symbol table .symtab and string table .strtab. Since everything in eBPF is addressed through file descriptors from user space, the tc front end first needs to generate the maps and based on the ELF's relocation entries, it inserts the file descriptor number into the related instructions as immediate value.

Depending on the pinning for the map, tc either fetches a map file descriptor from the bpf fs at the target location, or it generates a new map that was previously unpinned, and if requested, pins it. There are three different kind of scopes for sharing maps. They can be shared in a global namespace /sys/fs/bpf/tc/globals, in a object namespace /sys/fs/bpf/tc/<obj-sha>, or at a completely customized location. It allows eBPF maps to be shared between various cls_bpf instances. There are no restrictions with regards to sharing generic maps such as arrays or hash tables, even eBPF maps used by tracing eBPF programs (kprobes) can be shared with eBPF maps used by cls_bpf or act_bpf. When pinning, tc consults the ELF's symbol and string table to fetch a map name.

After having maps generated, tc looks up sections with program code, performs mentioned relocations with the generated map file descriptors and loads the program code into the kernel. When tail calls are used and tail called subsections are present in the ELF file, tc loads them into the kernel as well. They can have arbitrary nesting from a tc loader perspective, kernel runtime nesting is limited, of course. Also, the program's related program array for the eBPF tail calls can be pinned, which allows for later modifications from user space on the program's runtime behaviour. tc exec bpf has a graft option that will take care of replacing such sections during runtime. Grafting effectively undergoes the same procedure as loading a cls_bpf classifier initially, only that the resulting file descriptor is not pushed into the kernel through netlink, but rather through the corresponding map.

tc's cls_bpf front end also allows to pass generated map file descriptors through execvpe() over the environment to a new spawned shell, so programs can use it globally like stdin, stdout and stderr, or the file descriptor set can be passed via Unix domain sockets to another process. In both cases the cloned file descriptors lifetime is still tighly coupled with the process' lifetime. Retrieving new file descriptors through the bpf fs is the most flexible and preferred way [9], also for third party user space applications managing eBPF map contents.

Last but not least, the tc front end provides command line access for dumping the trace pipe via tc exec bpf dbg, where it automatically locates the mount point of trace fs.

Workflow

A typical workflow example for loading cls_bpf classifiers in da mode is straight forward. The generated object file from the foo.c program is called foo.o and contains two program sections p1 and p2. In the example, the foo.c file is first compiled with clang, the kernel eBPF JIT compiler enabled, then a clsact qdisc is added to device em1, the object file loaded as classifiers on ingress and egress path of em1, and eventually deleted again:

```
$ clang -O2 -target bpf -o foo.o -c foo.c

# sysctl -w net.core.bpf_jit_enable=1

# tc qdisc  add dev em1 clsact
# tc qdisc show dev em1
[...]
qdisc clsact ffff: parent ffff:fff1

# tc filter add dev em1 ingress bpf da \
  obj foo.o sec p1
# tc filter add dev em1 egress  bpf da \
  obj foo.o sec p2
```

```
# tc filter show dev em1 ingress
filter protocol all pref 49152 bpf
filter protocol all pref 49152 bpf handle
    0x1 foo.o:[p1] direct-action

# tc filter show dev em1 egress
filter protocol all pref 49152 bpf
filter protocol all pref 49152 bpf handle
    0x1 foo.o:[p2] direct-action

# tc filter del dev em1 ingress pref 49152
# tc filter del dev em1 egress  pref 49152
```

Programming

The `iproute2` repository ships with a couple of ”getting started” example files under `examples/bpf/` written in restricted C for running eBPF [1]. Implementing such classifiers is fairly straight forward. In contrast to a traditional user space C program, eBPF programs are restricted in a couple of ways. Every such classifier needs to be placed into ELF sections. Therefore, an object file can carry one or multiple eBPF classifiers.

They can share code among each other in two ways, either through `__always_inline` annotated functions, or commonly used tail call sections. The former is necessary as `clang` needs to compile the whole, flat program into a series of eBPF instructions residing self-contained in their related section.

Shared libraries or eBPF functions as relocation entries are not available. It is not possible for an eBPF loader like `tc` to ”stitch” them back together as a single flat array of eBPF instructions without redoing the job of a compiler. Thus, loaders have a ”contract” with `clang` which stipulates that the generated ELF file provides all necessary eBPF instructions contained in a given section. The only allowed relocation entries are in relation to maps, where file descriptors need to be set up first.

eBPF programs have a very limited stack space of 512 bytes, which needs to be taken into careful consideration when implementing them in C. Global variables as in typical C context are not possible, only eBPF maps (in `tc` it is `struct bpf_elf_map`) need to be defined in their own ELF section and can be referred by reference in program sections. If global ”variables” are needed, they can be realized, for example, as an eBPF per-CPU or non-per-CPU array map with a single entry, and referenced from various sections such as entry point sections, but also tail called sections.

Another restriction is that dynamic looping is not allowed in an eBPF program. Loops with compile-time known constant bounds can be used and unrolled with `clang`, though. Loops where the bounds cannot be determined during compilation time will get rejected by the verifier, since such programs are impossible to statically verify for guaranteed termination from all control flow paths.

Conclusion and Future Work

`cls_bpf` is a flexible and efficient classifier (as well as action) for the `tc` family. It allows for a high degree of data path programmability for a variety of different use cases involving parsing, lookup or updating (f.e. map state), and mangling network packets. When compiled by an underlying architecture's eBPF JIT back end, programs execute with native performance. Generally, eBPF has been designed to operate at environments that require high performance and flexibility at the same time.

While some of the internal details may appear complex, writing eBPF programs for `cls_bpf` is perhaps more comparable to some degree with user space application programming when certain constraints are taken into account. Handling `cls_bpf` front end from `tc` command line side involves only a few subcommands and is designed for ease of use.

The `cls_bpf` code, its `tc` front end, eBPF in general and its `clang` compiler back end are all open source, included in their related upstream projects, maintained and further developed by the upstream community.

There are a number of future enhancements and ideas still discussed and evaluated. Besides others, these include some form of offloading of `cls_bpf` into programmable NICs. The checkpoint restore in user space project (CRIU) is able to handle cBPF already, but eBPF support still needs to be implemented, which would be useful for container migrations.

References

[1] Begel, A.; Mccanne, S.; and Graham, S. L. 1999. Bpf+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *In SIGCOMM*, 123–134.

[2] Borkmann, D., and Sowa, H. F. 2014. net: bpf: make ebpf interpreter images read-only. Linux kernel, commit 60a3b2253c41.

[3] Borkmann, D., and Starovoitov, A. 2015. cls_bpf: introduce integrated actions. Linux kernel, commit 045efa82ff56.

[4] Borkmann, D.; Starovoitov, A.; and Sowa, H. F. 2015. bpf: add support for persistent maps/progs. Linux kernel, commit b2197755b263.

[5] Borkmann, D. 2013a. filter: bpf_asm: add minimal bpf asm tool. Linux kernel, commit 3f356385e8a4.

[6] Borkmann, D. 2013b. net: sched: cls_bpf: add bpf-based classifier. Linux kernel, commit 7d1d65cb84e1.

[7] Borkmann, D. 2015a. act_bpf: add initial ebpf support for actions. Linux kernel, commit a8cb5f556b56.

[8] Borkmann, D. 2015b. cls_bpf: add initial ebpf support for programmable classifiers. Linux kernel, commit e2e9b6541dd4.

[9] Borkmann, D. 2015c. {f,m}_bpf: allow for sharing maps. iproute2, commit 32e93fb7f66d.

[10] Borkmann, D. 2015d. tc: add ebpf support to f_bpf. iproute2, commit 11c39b5e98a1.

[1] `https://git.kernel.org/cgit/linux/kernel/git/shemminger/iproute2.git/tree/examples/bpf`

[11] Borkmann, D. 2015e. tc, bpf: finalize ebpf support for cls and act front-end. iproute2, commit 6256f8c9e45f.

[12] Borkmann, D. 2016. net, sched: add clsact qdisc. Linux kernel, commit 1f211a1b929c.

[13] de Bruijn, W. 2015a. packet: add classic bpf fanout mode. Linux kernel, commit 47dceb8ecdc1.

[14] de Bruijn, W. 2015b. packet: add extended bpf fanout mode. Linux kernel, commit f2e520956a1a.

[15] Drewry, W. 2012. seccomp: add system call filtering using bpf. Linux kernel, commit e2cfabdfd075.

[16] Gallek, C. 2016. soreuseport: setsockopt so_attach_reuseport_[ce]bpf. Linux kernel, commit 538950a1b752.

[17] Herbert, T. 2016. kcm: Kernel connection multiplexor module. Linux kernel, commit ab7ac4eb9832.

[18] Mccanne, S., and Jacobson, V. 1992. The bsd packet filter: A new architecture for user-level packet capture. 259–269.

[19] Pirko, J. 2012. team: add loadbalance mode. Linux kernel, commit 01d7f30a9f96.

[20] Pirko, J. 2015. tc: add bpf based action. Linux kernel, commit d23b8ad8ab23.

[21] Starovoitov, A., and Borkmann, D. 2014. net: filter: rework/optimize internal bpf interpreter's instruction set. Linux kernel, commit bd4cf0ed331a.

[22] Starovoitov, A. 2014a. bpf: expand bpf syscall with program load/unload. Linux kernel, commit 09756af46893.

[23] Starovoitov, A. 2014b. bpf: introduce bpf syscall and maps. Linux kernel, commit 99c55f7d47c0.

[24] Starovoitov, A. 2014c. bpf: verifier (add verifier core). Linux kernel, commit 17a5267067f3.

[25] Starovoitov, A. 2014d. net: filter: x86: internal bpf jit. Linux kernel, commit 622582786c9e.

[26] Starovoitov, A. 2015a. bpf: allow bpf programs to tail-call other bpf programs. Linux kernel, commit 04fd61ab36ec.

[27] Starovoitov, A. 2015b. tracing, perf: Implement bpf programs attached to kprobes. Linux kernel, commit 2541517c32be.

[28] Starovoitov, A. 2016. bpf: introduce bpf_map_type_stack_trace. Linux kernel, commit d5a3b1f69186.