NAME

BPF – BPF programmable classifier and actions for ingress/egress queueing disciplines

SYNOPSIS

eBPF classifier (filter) or action:

tc filter ... bpf [**object-file** OBJ_FILE] [**section** CLS_NAME] [**export** UDS_FILE] [**verbose**] [**direct-action** | **da**] [**skip_hw** | **skip_sw**] [**police** POLICE_SPEC] [**action** ACTION_SPEC] [**classid** CLASSID]

tc action ... bpf [**object-file** OBJ_FILE] [**section** CLS_NAME] [**export** UDS_FILE] [**verbose**]

cBPF classifier (filter) or action:

tc filter ... bpf [**bytecode-file** BPF_FILE | **bytecode** BPF_BYTECODE] [**police** POLICE_SPEC] [**action** ACTION_SPEC] [**classid** CLASSID]

tc action ... bpf [**bytecode-file** BPF_FILE | **bytecode** BPF_BYTECODE]

DESCRIPTION

Extended Berkeley Packet Filter (**eBPF**) and classic Berkeley Packet Filter (originally known as BPF, for better distinction referred to as **cBPF** here) are both available as a fully programmable and highly efficient classifier and actions. They both offer a minimal instruction set for implementing small programs which can safely be loaded into the kernel and thus executed in a tiny virtual machine from kernel space. An in-kernel verifier guarantees that a specified program always terminates and neither crashes nor leaks data from the kernel.

In Linux, it's generally considered that eBPF is the successor of cBPF. The kernel internally transforms cBPF expressions into eBPF expressions and executes the latter. Execution of them can be performed in an interpreter or at setup time, they can be just-in-time compiled (JIT'ed) to run as native machine code.

Currently, the eBPF JIT compiler is available for the following architectures:

* x86_64 (since Linux 3.18)
* arm64 (since Linux 3.18)
* s390 (since Linux 4.1)
* ppc64 (since Linux 4.8)
* sparc64 (since Linux 4.12)
* mips64 (since Linux 4.13)
* arm32 (since Linux 4.14)
* x86_32 (since Linux 4.18)

Whereas the following architectures have cBPF, but did not (yet) switch to eBPF JIT support:

* ppc32
* sparc32
* mips32

eBPF's instruction set has similar underlying principles as the cBPF instruction set, it however is modelled closer to the underlying architecture to better mimic native instruction sets with the aim to achieve a better run-time performance. It is designed to be JIT'ed with a one to one mapping, which can also open up the possibility for compilers to generate optimized eBPF code through an eBPF backend that performs almost as fast as natively compiled code. Given that LLVM provides such an eBPF backend, eBPF programs can therefore easily be programmed in a subset of the C language. Other than that, eBPF infrastructure also comes with a construct called "maps". eBPF maps are key/value stores that are shared between multiple eBPF programs, but also between eBPF programs and user space applications.

For the traffic control subsystem, classifier and actions that can be attached to ingress and egress qdiscs can be written in eBPF or cBPF. The advantage over other classifier and actions is that eBPF/cBPF provides the generic framework, while users can implement their highly specialized use cases efficiently. This means that the classifier or action written that way will not suffer from feature bloat, and can therefore execute its

task highly efficient. It allows for non-linear classification and even merging the action part into the classification. Combined with efficient eBPF map data structures, user space can push new policies like classids into the kernel without reloading a classifier, or it can gather statistics that are pushed into one map and use another one for dynamically load balancing traffic based on the determined load, just to provide a few examples.

PARAMETERS

object-file

points to an object file that has an executable and linkable format (ELF) and contains eBPF opcodes and eBPF map definitions. The LLVM compiler infrastructure with **clang(1)** as a C language front end is one project that supports emitting eBPF object files that can be passed to the eBPF classifier (more details in the **EXAMPLES** section). This option is mandatory when an eBPF classifier or action is to be loaded.

section

is the name of the ELF section from the object file, where the eBPF classifier or action resides. By default the section name for the classifier is called "classifier", and for the action "action". Given that a single object file can contain multiple classifier and actions, the corresponding section name needs to be specified, if it differs from the defaults.

export

points to a Unix domain socket file. In case the eBPF object file also contains a section named "maps" with eBPF map specifications, then the map file descriptors can be handed off via the Unix domain socket to an eBPF "agent" herding all descriptors after tc lifetime. This can be some third party application implementing the IPC counterpart for the import, that uses them for calling into **bpf(2)** system call to read out or update eBPF map data from user space, for example, for monitoring purposes or to push down new policies.

verbose

if set, it will dump the eBPF verifier output, even if loading the eBPF program was successful. By default, only on error, the verifier log is being emitted to the user.

direct-action | da

instructs eBPF classifier to not invoke external TC actions, instead use the TC actions return codes (**TC_ACT_OK**, **TC_ACT_SHOT** etc.) for classifiers.

skip_hw | skip_sw

hardware offload control flags. By default TC will try to offload filters to hardware if possible. **skip_hw** explicitly disables the attempt to offload. **skip_sw** forces the offload and disables running the eBPF program in the kernel. If hardware offload is not possible and this flag was set kernel will report an error and filter will not be installed at all.

police

is an optional parameter for an eBPF/cBPF classifier that specifies a police in **tc(1)** which is attached to the classifier, for example, on an ingress qdisc.

action

is an optional parameter for an eBPF/cBPF classifier that specifies a subsequent action in **tc(1)** which is attached to a classifier.

classid

flowid
provides the default traffic control class identifier for this eBPF/cBPF classifier. The default class identifier can also be overwritten by the return code of the eBPF/cBPF program. A default return code of **-1** specifies the here provided default class identifier to be used. A return code of the eBPF/cBPF program of 0 implies that no match took place, and a return code other than these two will override the default classid. This allows for efficient, non-linear classification with only a single eBPF/cBPF program as opposed to having multiple individual programs for various class identifiers which would need to reparse packet contents.

bytecode
is being used for loading cBPF classifier and actions only. The cBPF bytecode is directly passed as a text string in the form of **´s,c t f k,c t f k,c t f k,...´** , where **s** denotes the number of subsequent 4-tuples. One such 4-tuple consists of **c t f k** decimals, where **c** represents the cBPF opcode, **t** the jump true offset target, **f** the jump false offset target and **k** the immediate constant/literal. There are various tools that generate code in this loadable format, for example, **bpf_asm** that ships with the Linux kernel source tree under **tools/net/** , so it is certainly not expected to hack this by hand. The **bytecode** or **bytecode-file** option is mandatory when a cBPF classifier or action is to be loaded.

bytecode-file
also being used to load a cBPF classifier or action. It's effectively the same as **bytecode** only that the cBPF bytecode is not passed directly via command line, but rather resides in a text file.

EXAMPLES

eBPF TOOLING

A full blown example including eBPF agent code can be found inside the iproute2 source package under: **examples/bpf/**

As prerequisites, the kernel needs to have the eBPF system call namely **bpf(2)** enabled and ships with **cls_bpf** and **act_bpf** kernel modules for the traffic control subsystem. To enable eBPF/eBPF JIT support, depending which of the two the given architecture supports:

echo 1 > /proc/sys/net/core/bpf_jit_enable

A given restricted C file can be compiled via LLVM as:

clang -O2 -emit-llvm -c bpf.c -o - | llc -march=bpf -filetype=obj -o bpf.o

The compiler invocation might still simplify in future, so for now, it's quite handy to alias this construct in one way or another, for example:

```
__bcc() {
        clang -O2 -emit-llvm -c $1 -o - | \
        llc -march=bpf -filetype=obj -o "`basename $1 .c`.o"
}

alias bcc=__bcc
```

A minimal, stand-alone unit, which matches on all traffic with the default classid (return code of -1) looks like:

```
#include <linux/bpf.h>

#ifndef __section
```

```
# define __section(x)  __attribute__((section(x), used))
#endif

__section("classifier") int cls_main(struct __sk_buff *skb)
{
      return -1;
}

char __license[] __section("license") = "GPL";
```

More examples can be found further below in subsection **eBPF PROGRAMMING** as focus here will be on tooling.

There can be various other sections, for example, also for actions. Thus, an object file in eBPF can contain multiple entrance points. Always a specific entrance point, however, must be specified when configuring with tc. A license must be part of the restricted C code and the license string syntax is the same as with Linux kernel modules. The kernel reserves its right that some eBPF helper functions can be restricted to GPL compatible licenses only, and thus may reject a program from loading into the kernel when such a license mismatch occurs.

The resulting object file from the compilation can be inspected with the usual set of tools that also operate on normal object files, for example **objdump(1)** for inspecting ELF section headers:

```
objdump -h bpf.o
[...]
3 classifier    000007f8  0000000000000000  0000000000000000  00000040  2**3
                CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
4 action-mark   00000088  0000000000000000  0000000000000000  00000838  2**3
                CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
5 action-rand   00000098  0000000000000000  0000000000000000  000008c0  2**3
                CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
6 maps          00000030  0000000000000000  0000000000000000  00000958  2**2
                CONTENTS, ALLOC, LOAD, DATA
7 license       00000004  0000000000000000  0000000000000000  00000988  2**0
                CONTENTS, ALLOC, LOAD, DATA
[...]
```

Adding an eBPF classifier from an object file that contains a classifier in the default ELF section is trivial (note that instead of "object-file" also shortcuts such as "obj" can be used):

bcc bpf.c
tc filter add dev em1 parent 1: bpf obj bpf.o flowid 1:1

In case the classifier resides in ELF section "mycls", then that same command needs to be invoked as:

tc filter add dev em1 parent 1: bpf obj bpf.o sec mycls flowid 1:1

Dumping the classifier configuration will tell the location of the classifier, in other words that it's from object file "bpf.o" under section "mycls":

tc filter show dev em1
filter parent 1: protocol all pref 49152 bpf
filter parent 1: protocol all pref 49152 bpf handle 0x1 flowid 1:1 bpf.o:[mycls]

The same program can also be installed on ingress qdisc side as opposed to egress ...

tc qdisc add dev em1 handle ffff: ingress
tc filter add dev em1 parent ffff: bpf obj bpf.o sec mycls flowid ffff:1

... and again dumped from there:

tc filter show dev em1 parent ffff:
filter protocol all pref 49152 bpf
filter protocol all pref 49152 bpf handle 0x1 flowid ffff:1 bpf.o:[mycls]

Attaching a classifier and action on ingress has the restriction that it doesn't have an actual underlying queueing discipline. What ingress can do is to classify, mangle, redirect or drop packets. When queueing is required on ingress side, then ingress must redirect packets to the **ifb** device, otherwise policing can be used. Moreover, ingress can be used to have an early drop point of unwanted packets before they hit upper layers of the networking stack, perform network accounting with eBPF maps that could be shared with egress, or have an early mangle and/or redirection point to different networking devices.

Multiple eBPF actions and classifier can be placed into a single object file within various sections. In that case, non-default section names must be provided, which is the case for both actions in this example:

**tc filter add dev em1 parent 1: bpf obj bpf.o flowid 1:1 **
**action bpf obj bpf.o sec action-mark **
action bpf obj bpf.o sec action-rand ok

The advantage of this is that the classifier and the two actions can then share eBPF maps with each other, if implemented in the programs.

In order to access eBPF maps from user space beyond **tc(8)** setup lifetime, the ownership can be transferred to an eBPF agent via Unix domain sockets. There are two possibilities for implementing this:

1) implementation of an own eBPF agent that takes care of setting up the Unix domain socket and implementing the protocol that **tc(8)** dictates. A code example of this can be found inside the iproute2 source package under: **examples/bpf/**

2) use **tc exec** for transferring the eBPF map file descriptors through a Unix domain socket, and spawning an application such as **sh(1)** . This approach's advantage is that tc will place the file descriptors into the environment and thus make them available just like stdin, stdout, stderr file descriptors, meaning, in case user applications run from within this fd-owner shell, they can terminate and restart without losing eBPF maps file descriptors. Example invocation with the previous classifier and action mixture:

tc exec bpf imp /tmp/bpf
**tc filter add dev em1 parent 1: bpf obj bpf.o exp /tmp/bpf flowid 1:1 **
**action bpf obj bpf.o sec action-mark **
action bpf obj bpf.o sec action-rand ok

Assuming that eBPF maps are shared with classifier and actions, it's enough to export them once, for example, from within the classifier or action command. tc will setup all eBPF map file descriptors at the time when the object file is first parsed.

When a shell has been spawned, the environment will have a couple of eBPF related variables. BPF_NUM_MAPS provides the total number of maps that have been transferred over the Unix domain socket. BPF_MAP<X>'s value is the file descriptor number that can be accessed in eBPF agent applications, in other words, it can directly be used as the file descriptor value for the **bpf(2)** system call to retrieve

or alter eBPF map values. <X> denotes the identifier of the eBPF map. It corresponds to the **id** member of **struct bpf_elf_map** from the tc eBPF map specification.

The environment in this example looks as follows:

```
sh# env | grep BPF
  BPF_NUM_MAPS=3
  BPF_MAP1=6
  BPF_MAP0=5
  BPF_MAP2=7
sh# ls -la /proc/self/fd
  [...]
  lrwx------. 1 root root 64 Apr 14 16:46 5 -> anon_inode:bpf-map
  lrwx------. 1 root root 64 Apr 14 16:46 6 -> anon_inode:bpf-map
  lrwx------. 1 root root 64 Apr 14 16:46 7 -> anon_inode:bpf-map
sh# my_bpf_agent
```

eBPF agents are very useful in that they can prepopulate eBPF maps from user space, monitor statistics via maps and based on that feedback, for example, rewrite classids in eBPF map values during runtime. Given that eBPF agents are implemented as normal applications, they can also dynamically receive traffic control policies from external controllers and thus push them down into eBPF maps to dynamically adapt to network conditions. Moreover, eBPF maps can also be shared with other eBPF program types (e.g. tracing), thus very powerful combination can therefore be implemented.

eBPF PROGRAMMING

eBPF classifier and actions are being implemented in restricted C syntax (in future, there could additionally be new language frontends supported).

The header file **linux/bpf.h** provides eBPF helper functions that can be called from an eBPF program. This man page will only provide two minimal, stand-alone examples, have a look at **examples/bpf** from the iproute2 source package for a fully fledged flow dissector example to better demonstrate some of the possibilities with eBPF.

Supported 32 bit classifier return codes from the C program and their meanings:

0 , denotes a mismatch

-1 , denotes the default classid configured from the command line

else , everything else will override the default classid to provide a facility for non-linear matching

Supported 32 bit action return codes from the C program and their meanings (**linux/pkt_cls.h**):

TC_ACT_OK (0) , will terminate the packet processing pipeline and allows the packet to proceed

TC_ACT_SHOT (2) , will terminate the packet processing pipeline and drops the packet

TC_ACT_UNSPEC (-1) , will use the default action configured from tc (similarly as returning **-1** from a classifier)

TC_ACT_PIPE (3) , will iterate to the next action, if available

TC_ACT_RECLASSIFY (1) , will terminate the packet processing pipeline and start classification from the beginning

else , everything else is an unspecified return code

Both classifier and action return codes are supported in eBPF and cBPF programs.

To demonstrate restricted C syntax, a minimal toy classifier example is provided, which assumes that egress packets, for instance originating from a container, have previously been marked in interval [0, 255]. The program keeps statistics on different marks for user space and maps the classid to the root qdisc with the

marking itself as the minor handle:

```
#include <stdint.h>
#include <asm/types.h>

#include <linux/bpf.h>
#include <linux/pkt_sched.h>

#include "helpers.h"

struct tuple {
	long packets;
	long bytes;
};

#define BPF_MAP_ID_STATS	1 /* agent's map identifier */
#define BPF_MAX_MARK		256

struct bpf_elf_map __section("maps") map_stats = {
	.type		=	BPF_MAP_TYPE_ARRAY,
	.id		=	BPF_MAP_ID_STATS,
	.size_key	=	sizeof(uint32_t),
	.size_value	=	sizeof(struct tuple),
	.max_elem	=	BPF_MAX_MARK,
	.pinning	=	PIN_GLOBAL_NS,
};

static inline void cls_update_stats(const struct __sk_buff *skb,
				    uint32_t mark)
{
	struct tuple *tu;

	tu = bpf_map_lookup_elem(&map_stats, &mark);
	if (likely(tu)) {
		__sync_fetch_and_add(&tu->packets, 1);
		__sync_fetch_and_add(&tu->bytes, skb->len);
	}
}

__section("cls") int cls_main(struct __sk_buff *skb)
{
	uint32_t mark = skb->mark;

	if (unlikely(mark >= BPF_MAX_MARK))
		return 0;

	cls_update_stats(skb, mark);

	return TC_H_MAKE(TC_H_ROOT, mark);
}

char __license[] __section("license") = "GPL";
```

Another small example is a port redirector which demuxes destination port 80 into the interval [8080, 8087] steered by RSS, that can then be attached to ingress qdisc. The exercise of adding the egress counterpart and IPv6 support is left to the reader:

```
#include <asm/types.h>
#include <asm/byteorder.h>

#include <linux/bpf.h>
#include <linux/filter.h>
#include <linux/in.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <linux/tcp.h>

#include "helpers.h"

static inline void set_tcp_dport(struct __sk_buff *skb, int nh_off,
                                 __u16 old_port, __u16 new_port)
{
        bpf_l4_csum_replace(skb, nh_off + offsetof(struct tcphdr, check),
                            old_port, new_port, sizeof(new_port));
        bpf_skb_store_bytes(skb, nh_off + offsetof(struct tcphdr, dest),
                            &new_port, sizeof(new_port), 0);
}

static inline int lb_do_ipv4(struct __sk_buff *skb, int nh_off)
{
        __u16 dport, dport_new = 8080, off;
        __u8 ip_proto, ip_vl;

        ip_proto = load_byte(skb, nh_off +
                             offsetof(struct iphdr, protocol));
        if (ip_proto != IPPROTO_TCP)
                return 0;

        ip_vl = load_byte(skb, nh_off);
        if (likely(ip_vl == 0x45))
                nh_off += sizeof(struct iphdr);
        else
                nh_off += (ip_vl & 0xF) << 2;

        dport = load_half(skb, nh_off + offsetof(struct tcphdr, dest));
        if (dport != 80)
                return 0;

        off = skb->queue_mapping & 7;
        set_tcp_dport(skb, nh_off - BPF_LL_OFF, __constant_htons(80),
                      __cpu_to_be16(dport_new + off));
        return -1;
}

__section("lb") int lb_main(struct __sk_buff *skb)
{
```

```
        int ret = 0, nh_off = BPF_LL_OFF + ETH_HLEN;

        if (likely(skb->protocol == __constant_htons(ETH_P_IP)))
                ret = lb_do_ipv4(skb, nh_off);

        return ret;
}

char __license[] __section("license") = "GPL";
```

The related helper header file **helpers.h** in both examples was:

```
/* Misc helper macros. */
#define __section(x) __attribute__((section(x), used))
#define offsetof(x, y) __builtin_offsetof(x, y)
#define likely(x) __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)

/* Object pinning settings */
#define PIN_NONE      0
#define PIN_OBJECT_NS  1
#define PIN_GLOBAL_NS  2

/* ELF map definition */
struct bpf_elf_map {
   __u32 type;
   __u32 size_key;
   __u32 size_value;
   __u32 max_elem;
   __u32 flags;
   __u32 id;
   __u32 pinning;
   __u32 inner_id;
   __u32 inner_idx;
};

/* Some used BPF function calls. */
static int (*bpf_skb_store_bytes)(void *ctx, int off, void *from,
                                  int len, int flags) =
    (void *) BPF_FUNC_skb_store_bytes;
static int (*bpf_l4_csum_replace)(void *ctx, int off, int from,
                                  int to, int flags) =
    (void *) BPF_FUNC_l4_csum_replace;
static void *(*bpf_map_lookup_elem)(void *map, void *key) =
    (void *) BPF_FUNC_map_lookup_elem;

/* Some used BPF intrinsics. */
unsigned long long load_byte(void *skb, unsigned long long off)
  asm ("llvm.bpf.load.byte");
unsigned long long load_half(void *skb, unsigned long long off)
  asm ("llvm.bpf.load.half");
```

Best practice, we recommend to only have a single eBPF classifier loaded in tc and perform **all** necessary

matching and mangling from there instead of a list of individual classifier and separate actions. Just a single classifier tailored for a given use-case will be most efficient to run.

eBPF DEBUGGING

Both tc **filter** and **action** commands for **bpf** support an optional **verbose** parameter that can be used to inspect the eBPF verifier log. It is dumped by default in case of an error.

In case the eBPF/cBPF JIT compiler has been enabled, it can also be instructed to emit a debug output of the resulting opcode image into the kernel log, which can be read via **dmesg(1)** :

echo 2 > /proc/sys/net/core/bpf_jit_enable

The Linux kernel source tree ships additionally under **tools/net/** a small helper called **bpf_jit_disasm** that reads out the opcode image dump from the kernel log and dumps the resulting disassembly:

bpf_jit_disasm -o

Other than that, the Linux kernel also contains an extensive eBPF/cBPF test suite module called **test_bpf** . Upon ...

modprobe test_bpf

... it performs a diversity of test cases and dumps the results into the kernel log that can be inspected with **dmesg(1)** . The results can differ depending on whether the JIT compiler is enabled or not. In case of failed test cases, the module will fail to load. In such cases, we urge you to file a bug report to the related JIT authors, Linux kernel and networking mailing lists.

cBPF

Although we generally recommend switching to implementing **eBPF** classifier and actions, for the sake of completeness, a few words on how to program in cBPF will be lost here.

Likewise, the **bpf_jit_enable** switch can be enabled as mentioned already. Tooling such as **bpf_jit_disasm** is also independent whether eBPF or cBPF code is being loaded.

Unlike in eBPF, classifier and action are not implemented in restricted C, but rather in a minimal assembler-like language or with the help of other tooling.

The raw interface with tc takes opcodes directly. For example, the most minimal classifier matching on every packet resulting in the default classid of 1:1 looks like:

tc filter add dev em1 parent 1: bpf bytecode '1,6 0 0 4294967295,' flowid 1:1

The first decimal of the bytecode sequence denotes the number of subsequent 4-tuples of cBPF opcodes. As mentioned, such a 4-tuple consists of **c t f k** decimals, where **c** represents the cBPF opcode, **t** the jump true offset target, **f** the jump false offset target and **k** the immediate constant/literal. Here, this denotes an unconditional return from the program with immediate value of -1.

Thus, for egress classification, Willem de Bruijn implemented a minimal stand-alone helper tool under the GNU General Public License version 2 for **iptables(8)** BPF extension, which abuses the **libpcap** internal classic BPF compiler, his code derived here for usage with **tc(8)** :

```
#include <pcap.h>
#include <stdio.h>
```

```
int main(int argc, char **argv)
{
	struct bpf_program prog;
	struct bpf_insn *ins;
	int i, ret, dlt = DLT_RAW;

	if (argc < 2 || argc > 3)
		return 1;
	if (argc == 3) {
		dlt = pcap_datalink_name_to_val(argv[1]);
		if (dlt == -1)
			return 1;
	}

	ret = pcap_compile_nopcap(-1, dlt, &prog, argv[argc - 1],
				  1, PCAP_NETMASK_UNKNOWN);
	if (ret)
		return 1;

	printf("%d,", prog.bf_len);
	ins = prog.bf_insns;

	for (i = 0; i < prog.bf_len - 1; ++ins, ++i)
		printf("%u %u %u %u,", ins->code,
		       ins->jt, ins->jf, ins->k);
	printf("%u %u %u %u",
	       ins->code, ins->jt, ins->jf, ins->k);

	pcap_freecode(&prog);
	return 0;
}
```

Given this small helper, any **tcpdump(8)** filter expression can be abused as a classifier where a match will result in the default classid:

bpftool EN10MB 'tcp[tcpflags] & tcp-syn != 0' > /var/bpf/tcp-syn
tc filter add dev em1 parent 1: bpf bytecode-file /var/bpf/tcp-syn flowid 1:1

Basically, such a minimal generator is equivalent to:

tcpdump -iem1 -ddd 'tcp[tcpflags] & tcp-syn != 0' | tr '\n' ',' > /var/bpf/tcp-syn

Since **libpcap** does not support all Linux' specific cBPF extensions in its compiler, the Linux kernel also ships under **tools/net/** a minimal BPF assembler called **bpf_asm** for providing full control. For detailed syntax and semantics on implementing such programs by hand, see references under **FURTHER READING** .

Trivial toy example in **bpf_asm** for classifying IPv4/TCP packets, saved in a text file called **foobar** :

```
ldh [12]
jne #0x800, drop
ldb [23]
jneq #6, drop
```

ret #-1
drop: ret #0

Similarly, such a classifier can be loaded as:

bpf_asm foobar > /var/bpf/tcp-syn
tc filter add dev em1 parent 1: bpf bytecode-file /var/bpf/tcp-syn flowid 1:1

For BPF classifiers, the Linux kernel provides additionally under **tools/net/** a small BPF debugger called **bpf_dbg** , which can be used to test a classifier against pcap files, single-step or add various breakpoints into the classifier program and dump register contents during runtime.

Implementing an action in classic BPF is rather limited in the sense that packet mangling is not supported. Therefore, it's generally recommended to make the switch to eBPF, whenever possible.

FURTHER READING

Further and more technical details about the BPF architecture can be found in the Linux kernel source tree under **Documentation/networking/filter.txt** .

Further details on eBPF **tc(8)** examples can be found in the iproute2 source tree under **examples/bpf/** .

SEE ALSO

tc(8), **tc-ematch**(8) **bpf**(2) **bpf**(4)

AUTHORS

Manpage written by Daniel Borkmann.

Please report corrections or improvements to the Linux kernel networking mailing list: **<netdev@vger.kernel.org>**