Advanced programmability and recent updates with tc's cls_bpf.

Daniel Borkmann
Cisco
Switzerland
daniel@iogearbox.net

Abstract

With the introduction of eBPF into the Linux kernel and the added support for cls_bpf, tc has gained a highly programmable and efficient member in its repertoire of classifiers and actions which provides a generic and minimal bytecode language for tackling specific use-cases. Thanks to the LLVM back end for eBPF, programs can be written in a C-like language and compiled with clang into an object file that tc can load into the kernel for the cls_bpf back end. The set of just-in-time (JIT) compilers in the kernel translate eBPF instructions into machine-dependent instructions that allow for execution of programs with native performance. Since the last netdev conference, various new features related to cls_bpf have found its way into the Linux kernel. Therefore, this report can be regarded as a continuation of the netdev 1.1 paper on cls_bpf [14]. While the first paper was to provide an architectural overview, this paper discusses some of the recently introduced features for eBPF and cls_bpf in particular in more detail along with a few code examples.

Keywords

eBPF, cls_bpf, tc, programmable datapath, Linux kernel

Introduction

eBPF is a minimal, but highly flexible "virtual machine"-like construct in the Linux kernel which is used in a number of subsystems, most prominently networking and tracing [19]. It replaced the traditional in-kernel "classic" BPF (cBPF) interpreter, which is mostly known from tcpdump/libpcap filters that are passed as BPF bytecode into the kernel. Nowadays, the kernel speaks eBPF only and therefore, cBPF is translated into eBPF bytecode in kernel space before actual execution. eBPF consists of eleven 64 bit registers (r0-r10) with 32 bit sub-registers, a program counter and a eBPF stack space. Like cBPF, the instructions are 64 bit in size, and a few new instructions have been added such as load/store of double word, 64 bit ALU operations, a new call instruction, etc. The maximum instruction limit a program can carry is 4096 instructions, which is the same as in cBPF case. Next to forward jumps, also backward jumps are possible, but to a very limited degree where creation of loops is forbidden.

eBPF comes with a helper function concept that allows calls from an eBPF program into a well-defined set of kernel functions. Those kernel helper functions are a fixed part of the core kernel itself, tied to specific eBPF program types. New program types as well as helper functions cannot be added or extended through modular code, but must be accepted by the upstream community first. Some of the kernel helper functions are reserved for GPL-licensed BPF programs.

Besides helper functions, there is also the concept of maps that allows for keeping state across BPF program invocations. Maps typically act as efficient key/value store and can be shared arbitrarily among various eBPF programs, but also between eBPF programs and user space. There are various implementations of maps, such as arrays or hash tables, including per-CPU flavors of each. LLVM contains an eBPF back end, thus programs can be written in a C-like language, compiled by clang into object files, which various tools such as perf or tc can parse and load into the kernel.

In kernel space, the bytecode sequence is verified for safety, so that the kernel's operation cannot be harmed due to constructs like infinite loops, uninitialized memory, out of bounds accesses, pointer leakages or passing wrong types into helper functions. After verification, the kernel rewrites some of the passed BPF instructions, for example, to access data from the passed input context, which can be a skb in networking. Programs work with a limited shadowed structure that the kernel then needs to translate internally for actual access. After that phase, the instructions are JIT compiled by one of the currently available eBPF JIT back ends, such as x86_64, arm64, ppc64 or s390, so that the passed instructions can run with native performance.

Since work done in [5] and [2], cls_bpf has gained support for running eBPF as well, which makes cls_bpf a flexible and scalable choice as a programmable data plane from tc layer. Generic concepts and ideas from [25] are still preserved, that is, to provide a generic and flexible infrastructure to tackle specific use cases. cls_bpf can be integrated with the recently introduced sch_clsact pseudo qdisc [13] that allows for central ingress (__netif_receive_skb_core() and egress (__dev_queue_xmit()) hook points, and integration into classful qdiscs such as sch_htb as a usual classifier.

Recently Added Features

This section discusses recently introduced features for tc's cls_bpf programs since publication of the first part of the netdev paper in [14]. The discussed features are by no means a complete enumeration, but just illustrate some of the work that went into the upstream kernel for improving programmability and infrastructure around eBPF coupled with tc.

Tunneling and Encapsulation

One major feature that has been introduced recently is the ability to access tunneling protocols programmatically [32] [11] [10] from eBPF. Supported protocols are vxlan, geneve, gre, ipip. As back end infrastructure, they all use collect metadata mode which was introduced in [18]. The fundamental idea is that only a single net device is needed to represent multiple tunnels, which means that information about a particular tunnel must be passed to the related net device encapsulating the packet. This effort was initially done for OpenvSwitch (ovs) [18] to consolidate code between ovs and the rest of the kernel in order to switch to a pure net device based representation of ovs virtual ports, and to be able to scale with large number of tunnels, which was not possible for the existing net device based implementations without dedicated net devices for each configuration.

Since this infrastructure generalized from ovs side also fits to eBPF, helpers were added to get and set the generic BPF-based tunnel key representation as well as tunnel options. The structure that can be set and retrieved for BPF currently looks as follows:

```
struct bpf_tunnel_key {
  __u32 tunnel_id;
  union {
    __u32 remote_ipv4;
    __u32 remote_ipv6[4];
  };
  __u8 tunnel_tos;
  __u8 tunnel_ttl;
  __u16 tunnel_ext;
  __u32 tunnel_label;
};
```

From the BPF helpers bpf_skb_get_tunnel_key() and bpf_skb_set_tunnel_key(), the kernel maps the struct bpf_tunnel_key into a representation of struct ip_tunnel_info, which is used in tunneling back ends either to read out current settings of the given tunnel based on the packet header on receive, or to define them to fill the packet on transmit. The struct bpf_tunnel_key is kept rather generic on purpose, so that specific members are not tied to only one specific collect metadata back end. Due to uapi exposure, the kernel also implements a compatibility fixup around older struct bpf_tunnel_key representations. The most recent addition to support the collect metadata interface was done for ipip via [35] and [34], which now supports tunnels of type ipip, ipip6 and ip6ip6. The information is carried in a struct metadata_dst entry attached to the skb, which is just a normal dst entry, but with appended struct ip_tunnel_info accessed by the driver back end.

Tunnel options on the other hand does not have a fixed layout and extend the struct bpf_tunnel_key for allowing to pass down specific blobs for tunnel back ends. The eBPF helper interface is rather similar to that of tunnel keys, that is, bpf_skb_get_tunnel_opt() and bpf_skb_get_tunnel_opt(). Back ends that currently support passing tunnel options are vxlan and geneve. For the vxlan driver, this interface allows for setting and retrieving the group-based policy extension [29], whereas in geneve, TLV options [20] can be passed in a programmatic manner.

eBPF programs attached to cls_bpf can attach tunnel metadata and options as in the following example [36]:

```
struct vxlan_metadata {
  u32 gbp;
};

__section_cls_entry
int vxlan_set_tunnel(struct __sk_buff *skb)
{
  struct bpf_tunnel_key key = {};
  struct vxlan_metadata md;
  int ret;

  /* 172.16.1.100 */
  key.remote_ipv4 = 0xac100164;
  key.tunnel_id   = 2;
  key.tunnel_tos  = 0;
  key.tunnel_ttl  = 64;

  ret = bpf_skb_set_tunnel_key(skb, &key,
                               sizeof(key), 0);
  if (ret < 0)
    ...

  /* Set VXLAN Group Policy extension */
  md.gbp = 0x800FF;
  ret = bpf_skb_set_tunnel_opt(skb, &md,
                               sizeof(md));
  if (ret < 0)
    ...

  return TC_ACT_OK;
}
```

[36] provides examples for the receive part, but also demonstrates usage on other protocols like geneve including how TLVs are passed. In above case, the entries from the tunnel key and option are constants in the code, but they could just as well be derived based on other data, for example, coming from BPF maps shared with user space. Having them optimized as constants in the code becomes an option when, for example, programs are generated and compiled on the fly by higher level orchestration systems manging containers.

Direct Packet Access

A performance optimization called direct packet access was merged recently [33] [9] as well for cls_bpf (and also XDP program types), that allows reading and writing of skb data. Prior to that there existed two possibilities for reading skb data and one for writing, both came with their own advantages and disadvantages.

LLVM supports the following built-ins for its eBPF back end, that is, llvm.bpf.load.byte, llvm.bpf.load.half and llvm.bpf.load.word. They map to BPF_LD | BPD_ABS and BPF_LD | BPF_IND equivalents for BPF_B, BPF_H and BPF_W respectively, that have been carried over from cBPF mostly for legacy reasons in order to support efficient cBPF to eBPF migrations in the kernel, and as such they are the only skb-specific eBPF instructions. Based on the given offset, JITs can implement them quite efficiently, meaning, instructions are emitted that load from skb->data directly instead of emitting a function call. However, they need to call into a slow-path either if the accessed data is not within skb headlen range or if the passed offset is negative. For the former case, it is then required to walk skb fragmented data, which is quite expensive given that the load is only of size between a byte (BPF_B) up to a word (BPF_W) length. For the latter case when the offset is negative, the JIT compiler needs to emit a call to bpf_internal_load_pointer_neg_helper() that loads mentioned lengths relative to network header (SKF_NET_OFF) or relative to mac header (SKF_LL_OFF). In any case, the loaded data is stored in the target register in host endian order. The latter makes it rather cumbersome to work with protocols like IPv6, in particular since this kind of access is only limited to reading of data, thus for scenarios where address rewrites are necessary, further overhead of multiple data loads and endianess conversions back to network byte order are necessary.

This limitation was addressed later on by the bpf_skb_load_bytes() [4] helper. The helper can be regarded as a complementary addition to the bpf_skb_store_bytes() helper. It overcame the limitation that only up to 4 bytes could be loaded at once with the LLVM built-ins, so the new helper was made generic enough, that only the BPF stack space is the effective limitation for extracting data out of the skb, and therefore costs for the BPF helper call and bounds checks can be amortized. Optimizations to the verifier have been added in [12], [8] to educate the verifier that stack space memory does not need to be initialized when passing buffers to this helper, as the bpf_skb_load_bytes() is filling the buffers anyway with skb data. The only restriction added was that in case of errors, the uninitialized area must be zeroed by the helper. Since this is only relevant when passing wrong offsets and lengths, properly designed programs will never encounter such issue. The bpf_skb_load_bytes() helper stores the requested data area in network byte order and can also deal with non-linear skb data internally. Also, since JITs are designed to handle any BPF helper calls, no changes to JIT compilers were needed. As a result, bpf_skb_load_bytes() serves as a flexible alternative to the LLVM built-ins. The bpf_skb_store_bytes() helper works in a rather similar manner, only this time the programs pass the stack buffer space along with offset and length to the helper for storing into the skb. Furthermore, there is an option where the skb's hash can be invalidated or the checksum (for CHECKSUM_COMPLETE sums) be updated along the way. For packet checksums, the options of bpf_l3_csum_replace(), bpf_l4_csum_replace() and bpf_csum_diff() helpers exists.

While bpf_skb_load_bytes() and bpf_skb_store_bytes() helpers work quite well, further performance gain can be achieved by not needing to call helpers at all for loading and storing of skb data, and thus things like setup of registers for the helper calls, the call itself as well as bounds checking can be avoided altogether by making the verifier smarter while achieving similar functionality inline. [33] and [9] address this for read and write access by letting the verifier pattern match on tests that check accessible room and making sure both branches do not access beyond their probed bounds. One of the crucial aspects of this work is that neither JIT compilers nor the LLVM back end do need any changes to support this kind of access.

The idea of that work was to extend the shadow struct __sk_buff with data and data_end members, so that the verifier can convert them through normal context access into loading skb->data directly into a register, and a computed data_end pointer. The latter sits in the skb's control buffer coming after the struct qdisc_skb_cb control buffer for the tc layer. Since both members point into linear skb data, they are only valid as long as that underlying buffer is not changed, for example, due to reallocations with pskb_expand_head() for either uncloning a skb or pulling in non-linear data. Consequently, the verifier recognizes such helper function calls, since they are all listed in bpf_helper_changes_skb_data(). The latter helps JIT compilers to trigger emission of a reload of the skb->data that is cached in a temporary register. Moreover, it helps the verifier detecting that previous tests on data and data_end need to be invalidated.

The underlying mechanism is based on calling bpf_compute_data_end() before jumping into the BPF program as well as calling from helpers that change the skb's data, thus data_end is eventually always valid. The verifier as mentioned matches against data + X > data_end tests and analyzes both paths with regards to data accesses. For the case where data + X > data_end is indeed true, the verifier ensures that all further access is rejected. For the case where data + X > data_end is false, the verifier guarantees that all subsequent data accesses only happen within [0, X] range and reject any out of bounds attempts. Additionally, the verifier needs to track register contents which are derived from the register holding data, thus ALU operations need to be tracked for not accessing out of bounds as well. The logic also accounts for preventing possible arithmetical overflows, thus a maximum addressable range must not span beyond 0xffff.

Since bpf_compute_data_end() only assigns skb->data + skb_headlen(skb) to data_end, all access is limited to the linear data area of the skb, which means that any access to non-linear data would fail the data versus data_end check, and programs could bail out. To overcome this limitation, a new helper was introduced in [9] which can be called on such occasions in order to pull non-linear data into the linear data section of the skb. As this automatically invalidates previous bounds checks, the data versus data_end check has to be redone and would thus succeed.

The following example code demonstrates usage of direct read access for dropping pktgen-related frames on ingress:

```
static inline void *
skb_data(const struct __sk_buff *skb)
{
  return (void *)(long) skb->data;
}

static inline void *
skb_data_end(const struct __sk_buff *skb)
{
  return (void *)(long) skb->data_end;
}

static inline const int skb_room(void)
{
  return sizeof(struct eth_hdr) +
         sizeof(struct iphdr) +
         sizeof(struct udphdr);
}

__section_cls_entry
int dropper_main(struct __sk_buff *skb)
{
  void *data_end = skb_data_end(skb);
  void *data = skb_data(skb);
  struct eth_hdr *eth;
  struct udphdr *udp;
  struct iphdr *iph;

  if (data + skb_room() > data_end) {
    if (bpf_skb_pull_data(skb, skb_room()))
      return TC_ACT_OK;
    if (data + skb_needed_room() > data_end)
      return TC_ACT_OK;
  }
```

```
  eth = data;
  if (eth->h_proto != htons(ETH_P_IP))
    return TC_ACT_OK;
  iph = data + sizeof(*eth);
  if (iph->protocol != IPPROTO_UDP ||
      iph->ihl != 5)
    return TC_ACT_OK;
  if (ip_is_fragment(iph))
    return TC_ACT_OK;
  udp = data + sizeof(*eth) + sizeof(*iph);
  if (udp->dest == htons(PKTGEN_UDP_PORT))
    return TC_ACT_SHOT;
  return TC_ACT_OK;
}
```

If known based on NIC/driver characteristics that the initial data versus data_end check never fails due to the fact that enough header space was pulled in, the bpf_skb_pull_data() with the second test can then be omitted, of course.

The direct write part works in a similar way, but is slightly more complicated since the invariant needs to be ensured that during program runtime the skb stays uncloned. Therefore once writes are detected by the verifier, a prologue is added to the BPF instruction sequence which checks as a heuristic that skb->cloned flag is set and if so, performs a bpf_skb_pull_data() call for the entire head length to effectively unclone the skb. Similarly, the bpf_clone_redirect() helper must unclone the original skb from the spawned cloned one. Extra helpers for adjusting the checksum (bpf_csum_update()) and for clearing the hash (bpf_set_hash_invalid()) were added that can each be called once after all mangling was performed. Both kind of direct accesses can reduce overhead for programs performing actions like parsing and packet rewriting.

Along with [9] comes also the possibility to directly access packet data from cls_bpf for helper functions, which helps to optimize use-cases like the ILA data plane written in eBPF as proposed in [37], [38]. The idea is that instead of copying packet data to the eBPF stack first and then passing the stack buffer to the helper, the extra copy can be avoided by allowing the packet data directly. Map helper functions can make use of this as well as bpf_csum_diff().

Event Notifications

One of the recently added features to eBPF programs for cls_bpf is the ability to push event notifications up to user space [3]. This feature is useful in a number of ways, for example, for sampling, monitoring or logging of packet data or state, debugging of BPF programs in general and pushing wake-up events to management daemons in user space that control the BPF data plane. The idea is very similar to [31], where the perf event array can be reused. The push into the event-pipe is limited to only be done unidirectional from kernel to user space, but not vice versa.

Similarly as in [31], a high-performance per-CPU mmap(2)ed event ring buffer is utilized from the perf infrastructure to collect incoming events and for triggering the user process wake-ups. Wake-ups from poll(2) can be defined whether they should i) never happen, so the process itself would be busy-polling the rings, ii) be triggered after each or a specified number of events, iii) be triggered after the ring buffer has been filled up to a watermark with a given number of bytes.

For debugging purposes, this infrastructure might be preferred over bpf_trace_printk() when a high rate of events need to be dealt with, since it is more efficient due to not needing to prepare a printk()-like string via __trace_printk() facility.

Thus, a program is flexible enough to define their own structure layout for a ring buffer slot, which means over time layouts can be changed since not being part of the uapi. The ring buffer itself is lockless, which allows for high event rates. In case the user space consumer is not fast enough to process events and thus allow kernel side to move on by updating data_tail pointer, the number of lost events are recorded as well in a ring buffer slot, so that on next query, the daemon processing events can act accordingly.

```
struct bpf_elf_map __section_maps tc_events = {
  .type       = BPF_MAP_TYPE_PERF_EVENT_ARRAY,
  .size_key   = sizeof(int),
  .size_value = sizeof(int),
  .pinning    = PIN_GLOBAL_NS,
  .max_elem   = __NR_CPUS__,
};

__section_cls_entry
int simple_event(struct __sk_buff *skb)
{
  struct foo {
    u32 mark;
    ...
  } data = {
    .mark = skb->mark,
    ...
  };
  u64 plen = 64;

  bpf_event_output(skb, &tc_events, plen << 32 |
                   BPF_F_CURRENT_CPU,
                   &data, sizeof(data));
  return TC_ACT_OK;
}
```

In above example, we can define some structure on the stack and pass it to the bpf_event_output() helper. One can either pass a specific map index or BPF_F_CURRENT_CPU flag to use the perf event map at the index of the current CPU number. The corresponding perf event handler must be pinned to the current CPU for further processing. Such a perf event map can have multiple users attached at various indices. Follow-up work [15] [7] optimized the helper further to avoid an extra stack copy for skb data, thus for the perf event ring buffer's raw records, support for fragmented data has been added so that the passed stack buffer can be transferred along with some payload data as a sample.

One heavy user of this facility is, for example, cilium [16], which provides container networking based on BPF through tc and cls_bpf. A whole packet tracing facility has been implemented around this helper that marks skbs going through BPF programs generated from cilium that implement functionality such as NAT64, scalable network policy or load balancers for containers.

JITs, Offloads and Hardening

For accelerating eBPF program execution, a number of architectures implement a JIT compiler, which translates an eBPF program into an executable opcode image that can be jumped into natively.

Current JITs today that have complete or mostly complete eBPF support are x86_64, arm64, s390 and most recent addition was ppc64 [26]. arm64 is currently missing atomic add (BPF_XADD) support for word and double words so far [28], and ppc64 does not have support for set_memory_ro() and set_memory_rw().

The latter will not create any operability issues regarding eBPF features, but it would be desirable if such executable pages could be locked down as read-only, too. Both set_memory_* helpers are supported through CONFG_DEBUG_SET_MODULE_RONX, which currently appears to be more of a second class debugging citizen, but with the help of kernel hardening project this might change [17]. The same read-only lockdown also happens for eBPF interpreter programs during their whole lifetime [1].

As another hardening measure, eBPF JIT images have a randomized start address with a gap filled with trap instructions. Reason is that with the help of other possible kernel bugs, an attacker could spray a large enough number of JITed BPF programs into kernel space, where the constants that are part of the user-controlled BPF instruction sequence could contain actual CPU opcodes themselves. Crafted in a way, so that this would still pass the kernel verifier, the CPU, while still in kernel mode, could jump into such an interleaved location where it would then start to execute such opcodes passed in through constants.

This additionally requires kernel bugs from elsewhere, which would then need to be able to trigger a jump into one of the loaded programs. A proof-of-concept with regards to spraying was presented by McAllister [24] in 2012, where the kernel has been sprayed with BPF programs attached as unprivileged socket filters. The file descriptors of these sockets have been placed into a Unix domain socket through `SCM_RIGHTS`, which means that while the user space application can create and close many of such sockets, the kernel needs to keep such file descriptors alive for other processes to pick these up and thus they need to be maintained in kernel space, including the BPF program. By using a tree-like structure with `AF_UNIX` socketpairs, a fairly sufficient number of BPF programs were sprayed into the kernel. Back then those JITed programs allocated with `module_alloc()` were starting at the beginning of a page, thus that with enough programs loaded and an reduced address search space, chances of guessing were reduced by McAllister up to one in a fifty to make the right jump [24].

An example of such code injection from [24] by abusing the `BPF_LD | BPF_IMM` instruction looks like:

```
Emit a 3-byte x86 instruction, embedded
within a BPF "load immediate". The most
significant byte of the loaded quantity
is 0xa8.

The kernel's BPF JIT compiles a sequence
of such instructions into:

  b8 XX YY ZZ a8    mov $0xa8ZZYYXX, %eax
  b8 PP QQ RR a8    mov $0xa8RRQQPP, %eax
  b8 [...]

Jumping one byte into this code produces
an instruction stream like:

  XX YY ZZ          payload instruction
  a8 b8             test $0xb8, %al
  PP QQ RR          payload instruction
  a8 b8             test $0xb8, %al
  [...]
```

As a result, a randomized start address with a trap section helps to make it a bit harder, but as recently shown by Reshetova [27], it does not provide full protection when BPF programs cross page boundaries and the injected code being improved with `nop` instructions to make the jump into the opcodes more likely to execute the crafted payload, because we once again have parts of the program at a page start address again.

Moreover, besides cBPF programs, eBPF programs come with a load instruction for 64 bit constants, which would result in further increasing injection possibilities. Since [30] these can also be run by unprivileged users through socket filters.

To mitigate these issues, a generic constant blinding facility has been developed [6]. The basic idea of this is that constants are blinded out when generating the JIT image by rewriting the raw constant with an xored pseudo-random number, which gets loaded as such into a helper register. That register is then being xored again with only the pseudo-random number used before, so that the original raw constant is now residing in that helper register, and finally the original BPF instruction is rewritten from being a immediate-based into a register-based operation.

For example both `mov` operations below are replaced by a `mov-xor-mov` sequence:

```
echo 0 > /proc/sys/net/core/bpf_jit_harden

  ffffffffa034f5e9 + <x>:
  [...]
  39:   mov    $0xa8909090,%eax
  3e:   mov    $0xa8909090,%eax
  [...]

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

  ffffffffa034f1e5 + <x>:
  [...]
  39:   mov    $0xe1192563,%r10d
  3f:   xor    $0x4989b5f3,%r10d
  46:   mov    %r10d,%eax
  49:   mov    $0xb8296d93,%r10d
  4f:   xor    $0x10b9fd03,%r10d
  56:   mov    %r10d,%eax
  [...]
```

The blinding functionality itself was implemented in a way that does not require low-level JIT changes, but instead is performed on BPF bytecode level which is also easier to maintain. Thus when JITing phase starts, a JIT compiler calls into related blinding helpers and creates a clone of the program that is then blinded. While the JIT compiler then tries to JIT the blinded BPF instruction sequence, a fallback to the unblinded sequence is performed in case an error (f.e. memory allocation) occurred during JITing process. That way, the eBPF interpreter can continue with the unblinded image and also does not need an additional helper register for these operations. The integration of this for JITs in the most straight forward way is to just map the `BPF_REG_AX` into an unused temporary register.

The `/proc/sys/net/core/bpf_jit_harden` sysctl switch added along with this infrastructure comes in three operating modes: 0 - do not blind, 1 - blind load of unprivileged programs, 2 - blind all programs. While the latter is useful for testing, the normal operating mode ensures that blinding on privileged programs has zero performance overhead. The other advantage resulting from generic blinding is that for those architectures that have an eBPF JIT, the constant blinding takes also effect for cBPF programs, since cBPF programs are migrated to eBPF in the kernel anyway. Also, the performance overhead varies depending on how many instructions are used along with immediates. It however still provides significant performance benefits compared to execution via interpreter [6].

With regards to offloading `cls_bpf` programs to the NIC, Netronome recently became the first vendor that supports offloading of eBPF instructions to their smart NICs with the help of a JIT compiler [22] [23] that translates into instructions for their programmable NFP engines. Extensive details of the design and implementation are published in [21] and therefore out of scope for this report.

Conclusion and Future Work

As it can be seen from the recently introduced features for `cls_bpf` programs, the infrastructure around BPF is constantly improved and optimized for efficiency. There are however a lot of challenges to tackle in near to mid-term future. Some of them are further discussed in this section.

One of them is to add support for encryption in terms of MACsec and IPsec, so that tunneled traffic can also be secured against potential adversaries. One of the ideas at the moment is to add a similar interface we have with collect metadata discussed in this context for tunneling, but generic enough to accommodate various crypto back ends.

There is currently also no sufficient support for IPv4/IPv6 fragmentation handling, thus work in this area is needed to make better use of existing kernel facilities we have for dealing with fragmentation.

Usability and documentation around eBPF in general needs more work to lower entrance barriers for writing programs. One aspect that falls into this context as well is that for more complex programs, verifier error logs can quickly become quite verbose as the verifier walks all possible program paths to check for safety, which makes it hard to find bugs in programs. Perhaps a new logging facility needs to be designed that makes tracing issues from static analysis easier to resolve. For example, it would be desirable, perhaps with the help of LLVM, to annotate verifier complaints back to the original source code location causing a particular issue.

References

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

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

[3] Borkmann, D., and Starovoitov, A. 2016. bpf: add event output helper for notifications/sampling/logging. Linux kernel, commit bd570ff970a5.

[4] Borkmann, D. 2015a. bpf: add bpf_skb_load_bytes helper. Linux kernel, commit 05c74e5e53f6.

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

[6] Borkmann, D. 2016a. bpf: add generic constant blinding for use in jits. Linux kernel, commit 4f3446bb809f.

[7] Borkmann, D. 2016b. bpf: avoid stack copy and use skb ctx for event output. Linux kernel, commit 555c8a8623a3.

[8] Borkmann, D. 2016c. bpf: convert relevant helper args to arg_ptr_to_raw_stack. Linux kernel, commit 074f528eed40.

[9] Borkmann, D. 2016d. bpf: direct packet write and access for helpers for clsact progs. Linux kernel, commit 36bbef52c7eb.

[10] Borkmann, D. 2016e. bpf: support for access to tunnel options. Linux kernel, commit 14ca0751c96f.

[11] Borkmann, D. 2016f. bpf: support ipv6 for bpf_skb_{set,get}_tunnel_key. Linux kernel, commit c6c33454072f.

[12] Borkmann, D. 2016g. bpf, verifier: add arg_ptr_to_raw_stack type. Linux kernel, commit 435faee1aae9.

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

[14] Borkmann, D. 2016i. On getting tc classifier fully programmable with cls_bpf. Proceedings of netdev 1.1, Feb 10-12, 2016, Seville, Spain.

[15] Borkmann, D. 2016j. perf, events: add non-linear data support for raw records. Linux kernel, commit 7e3f977edd0b.

[16] Cilium Authors, V. 2016. Cilium - bpf & xdp for containers. https://github.com/cilium/cilium.

[17] Cook, K. 2016. Kernel self-protection. Linux kernel, Documentation/security/self-protection.txt.

[18] Graf, T. 2016. Lightweight & flow based encapsulation. Linux kernel, https://lwn.net/Articles/651497/.

[19] Gregg, B. 2015. ebpf: One small step. http://www.brendangregg.com/blog/2015-05-15/ebpf-one-small-step.html.

[20] Gross, J.; Ganga, I.; and Sridhar, T. 2016. Geneve: Generic network virtualization encapsulation. IETF draft, https://tools.ietf.org/html/draft-ietf-nvo3-geneve-03.

[21] Kicinski, J., and Viljoen, N. 2016. ebpf/xdp hardware offload to smartnics. Proceedings of netdev 1.2, Oct 5-7, 2016, Tokyo, Japan.

[22] Kicinski, J. 2016a. nfp: add bpf to nfp code translator. Linux kernel, commit cd7df56ed3e6.

[23] Kicinski, J. 2016b. nfp: bpf: add hardware bpf offload. Linux kernel, commit 7533fdc0f77f.

[24] McAllister, K. 2012. Attacking hardened linux systems with kernel jit spraying. http://mainisusuallyafunction.blogspot.com/2012/11/attacking-hardened-linux-systems-with.html.

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

[26] N. Rao, N. 2016. powerpc/ebpf/jit: Implement jit compiler for extended bpf. Linux kernel, commit 156d0e290e96.

[27] Reshetova, E. 2016. Bpf jit spray attack - proof of concept code for modern kernel. http://www.openwall.com/lists/kernel-hardening/2016/05/03/5.

[28] Shen Lim, Z. 2016. Arm64 ebpf jit todo list. https://github.com/zlim/bpf#todo-arm64-ebpf.

[29] Smith, M., and Kreeger, L. 2016. Vxlan group policy option. IETF draft, https://tools.ietf.org/html/draft-smith-vxlan-group-policy-02.

[30] Starovoitov, A. 2015a. bpf: enable non-root ebpf programs. Linux kernel, commit 1be7f75d1668.

[31] Starovoitov, A. 2015b. bpf: introduce bpf_perf_event_output() helper. Linux kernel, commit a43eec304259.

[32] Starovoitov, A. 2016a. bpf: add helpers to access tunnel metadata. Linux kernel, commit d3aa45ce6b94.

[33] Starovoitov, A. 2016b. bpf: direct packet access. Linux kernel, commit 969bf05eb3ce.

[34] Starovoitov, A. 2016c. ip6_tunnel: add collect_md mode to ipv6 tunnels. Linux kernel, commit 8d79266bc48c.

[35] Starovoitov, A. 2016d. ip_tunnel: add collect_md mode to ipip tunnel. Linux kernel, commit cfc7381b3002.

[36] Tu, W. 2016. samples/bpf: Add tunnel set/get tests. Linux kernel, commit 6afb1e28b859.

[37] Yue, A. 2016a. samples/bpf: ilarouter for tc. http://patchwork.ozlabs.org/patch/674160/.

[38] Yue, A. 2016b. samples/bpf: ilarouter for xdp. http://patchwork.ozlabs.org/patch/674159/.