XDP-Programmable Data Path in the Linux Kernel

DIPTANU GON CHOUDHURY

Diptanu Gon Choudhury works on large-scale distributed systems. His interests lie in designing large-scale cluster schedulers, highly available control plane systems, and high-performance network services. He is the author of the upcoming O'Reilly book *Scaling Microservices*.
diptanuc@gmail.com

Berkeley Packet Filter was introduced almost two decades ago and has been an important component in the networking subsystem of the kernel for assisting with packet filtering. Extended BPF can do much more than that and is gradually finding its way into more kernel subsystems as a generic event-processing infrastructure. In this article, I provide enough background to help you understand how eBPF works, then describe a simple and fast firewall using Express Data Path (XDP) and eBPF.

Berkeley Packet Filter (BPF) has been around for more than two decades, born out of the requirement for fast and flexible packet filtering machinery to replace the early '90s implementations, which were no longer suitable for emerging processors. BPF has since made its way into Linux and BSDs via libpcap, which is the foundation of tcpdump.

Instead of writing the packet filtering subsystem as a kernel module, which can be unsafe and fragile, McCanne and Jacobson designed an efficient yet minimal virtual machine in the kernel, which allows execution of bytecode in the data path of the networking stack.

The virtual machine was very simple in design, providing a minimalistic RISC-based instruction set, with two 32-bit registers, but it was very effective in allowing developers to express logic around packet filtering. BPF owes its relative longevity to two factors—flexibility and performance. The design goal was to design the subsystem in a protocol-agnostic manner and the instruction set to be able to handle unforeseen use cases.

A sample BPF program that filters every IP packet:

```
(000) ldh     [12]
(001) jeq     #0x800        jt 2      jf 3
(002) ret     #262144
(003) ret     #0
```

This program loads a half-word from offset 12, checks if the value is #0x800, and returns true if it matches and false if it doesn't.

The flexible instruction set allowed programmers to use BPF for all sorts of use cases such as implementing packet filtering logic for iptables, which performs very well under high load and allows for more complex filtering logic. Having a protocol-independent instruction set allowed developers to update these filters without writing kernel modules; having a virtual machine run the instructions provided a secure environment for execution of the filters. A significant milestone was reached in 2011 when a just in time (JIT) compiler was added to the kernel, which allowed translating BPF bytecode into the host system's assembly instruction set. However, it was limited to only x86_64 architecture because every instruction was mapped one on one to an x86 instruction or register.

Things took an interesting turn when the BPF subsystem was "extended" in the Linux operating system in 2013, and since then BPF is used in a lot more places, including tracing and security subsystems, besides networking.

Extended BPF

Linux 3.18 had the first implementation of extended BPF (eBPF), which made significant improvements from its precursor. While the original BPF virtual machine had two 32-bit registers, eBPF had 10 64-bit registers, added more instructions that were close to the hardware, and made it possible to call a subset of the kernel functions. All the BPF registers matched with the actual hardware registers, and BPF's calling conventions were similar to the Linux kernel's ABI in most architectures. One of the important outcomes of this was that it was now possible to use a compiler like LLVM to emit BPF bytecode from a subset of the C programming language. Another important addition was the BPF_CALL instruction, which allows BPF programs to call helper functions from the kernel allowing reuse of certain existing kernel infrastructure.

The important point to keep in mind is that eBPF today can be used as a general-purpose event-processing system by various subsystems in the kernel. These events can come from various different sources such as a kprobe tracepoint or an arrival of a packet in the receive queue of the network driver. Support for BPF has gradually been added to various strategic points in the kernel such that when code in those kernel subsystems execute, the BPF programs are triggered. The kernel subsystems that trigger a BPF program dictate the capability of a BPF program, and usually every BPF program type is connected to a kernel subsystem. For example, the traffic control subsystem supports the BPF_PROG_TYPE_SCHED_CLS and BPF_PROG_TYPE_SCHED_ACT program types that allow developers to write BPF to classify traffic and control behavior of the traffic classifier actions, respectively. Similarly, the `seccomp` subsystem can invoke a BPF program to determine whether a userspace process can make a particular syscall.

Writing the BPF bytecode for anything nontrivial can be challenging, but things have become a lot simpler since BPF has been added as a target in LLVM and users can now generate BPF in a subset of the C programming language.

In today's Linux kernel, the old BPF instruction set, commonly known as cBPF, is transparently translated to eBPF instructions. *I will use eBPF and BPF interchangeably from here on.*

BPF Maps

An introduction to BPF is incomplete without discussing BPF maps. BPF programs by themselves are stateless, and so maps allow programs to maintain state between invocations. For example, we could write a BPF program that prints a trace message whenever the `inet_listen` function is called in the kernel. However, if we wanted to expose that information as a counter to some monitoring tool, we would need the program to maintain state somewhere and increment a counter every time the method is called. This is where BPF maps come in. BPF maps are generic data structures implemented in the kernel where eBPF programs can store arbitrary data. These data structures, commonly referred to as maps, treat the data as opaque, and hence programs can store arbitrary bytes as key-value as appropriate. Maps can only be created or deleted from the userspace; BPF programs access the maps by using helper functions such as `bpf_map_lookup_elem`.

As of this writing, there are 11 different types of maps implemented in the kernel today, some of them generic and others used specifically with helper functions. The generic maps are:

```
BPF_MAP_TYPE_HASH
BPF_MAP_TYPE_ARRAY
BPF_MAP_TYPE_PERCPU_HASH
BPF_MAP_TYPE_PERCPU_ARRAY
BPF_MAP_TYPE_LRU_HASH
BPF_MAP_TYPE_LRU_PERCPU_HASH
BPF_MAP_TYPE_LPM_TRIE
```

Each of them is designed for a specific use case, so it's useful to understand the performance characteristics and their heuristics before starting to use them in BPF programs. For example, if we were designing a filter that increments a counter for every UDP packet that is being dropped, it would be best to use a per-CPU hash map so that the counters can be incremented without any synchronization to prevent multiple instances of the BPF program being triggered on different CPUs simultaneously. The non-generic maps are best described in the context of the documentation for the operations with which they can be used.

The BPF Syscall

The BPF syscall introduced in kernel 3.18 is the main workhorse for userspace programs to interact with the BPF infrastructure. The syscall multiplexes almost all the operations that userspace processes need to perform when handling BPF programs and maps. The syscall's usage includes, but is not limited to, loading BPF filters into the kernel, creating new maps, or retrieving data from existing ones.

```
int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

- `cmd`—It could be one of the many operations that the syscall can perform. There are 10 such commands in total, six of which are documented in the man page.
- `attr`—A union that provides context to the command. For example, when used with the BPF_PROG_LOAD command, it allows the bytecode to be passed to the kernel, and with the BPF_MAP_CREATE, it lets the user define the size of the key and values of the map.
- `size`—The size of the attr union.

The syscall returns 0 when it succeeds in most cases, except for BPF_PROG_LOAD and BPF_MAP_CREATE, which return the file descriptor of the BPF object created. For any failures, it returns -1 and sets the appropriate errno.

However, most userspace programs don't use the raw syscalls; the BPF Compiler Collection (BCC) provides the libbpf library, which has some wrapper functions that make working with BPF objects easier.

```
int bpf_prog_load(enum bpf_prog_type prog_type, const char
*name, const struct bpf_insn *insns, int prog_len, const char
*license, unsigned kern_version, int log_level, char *log_buf,
unsigned log_buf_size)
```

For example, the above wrapper function creates the attribute union for the `bpf()` syscall and wires in appropriate parameters. The kernel samples include good examples of usage of the libbpf library and other userspace helpers.

BPF Verifier

Safety is a very important concern for BPF programs, especially because they tend to run in the performance-critical sections of the kernel. The BPF infrastructure includes an in-kernel verifier that uses CFG (control flow graph) to determine that the BPF program terminates within the limit of maximum number of instructions. The verifier, for example, forbids loops and makes sure that maps are not destroyed until a program that uses it doesn't terminate. The verifier also statically ensures the safety of the calls to the helper functions by checking that the types of the data in the BPF VM register matches with the types of the helper function arguments.

In addition to ensuring type safety, the verifier also ensures safety of the program by prohibiting out of bounds jumps and out-of-range data access. The verifier also restricts what kernel functions and which data structures can be accessed based on the BPF program type.

BPF File System

BPF maps and filters are effectively kernel resources exposed to userspace via file descriptors backed by anonymous inodes; this comes with some benefits but also interesting challenges. Once a userspace program exits, the BPF program would get destroyed, and so would the maps related to that program. The lifetime of a BPF program and maps are tied to that of the userspace process that loaded the program, which prevents maps from persisting between filter invocations. As a result, maintaining state between program invocations becomes impossible. To overcome these limitations, the BPF infrastructure comes with a file system where BPF objects like maps and programs can be pinned to a path in the file system. This process is commonly known as Object Pinning, and two new commands, BPF_OBJ_PIN and BPF_OBJ_GET, facilitate pinning and retrieving an existing pinned object. The command simply needs file descriptors and the path to which the object is going to be pinned.

An interesting aspect of BPF objects being exposed as file system objects is that processes with higher privileges could create the objects and pin them to the file system and then drop their permission. For example, this allows lower-privileged userspace tools, like monitoring tools, to read telemetry data from maps.

BPF Tail Calls

BPF programs are limited to 4096 instructions, but it's possible to chain multiple programs together via *tail calls*. This technique allows a BPF program to call another BPF program when it finishes. Tail calls are implemented by long jumps inside the VM, which reuse the same stack frame. BPF tail calls are different from normal functions in the sense that once the new function is invoked when the current function ends, the previous program ends. Data could be shared between stages by using per-CPU maps as temporary buffers. Tail calls are used to modularize BPF programs. For example, a program could parse the headers of a network packet, and the following program could implement some other logic like tracing or running classifier actions based on the headers.

There are certain limitations to tail calls:

1. Only similar programs can be chained together.
2. The maximum number of tail calls allowed is 32.
3. Programs which are JITed can't be mixed with the ones that are not JITed.

As stated earlier, various kernel subsystems now have support for BPF. I will cover one such area that is part of the networking subsystem.

Express Data Path (XDP)

The networking subsystem of the kernel is one of the more performance-sensitive areas—there is always ongoing work to improve performance! Over the years userspace networking frameworks like DPDK have attracted users with the promise of faster packet processing by bypassing the kernel network stack. While it's lucrative for userspace programs to get access to network devices and improve on some data copies by bypassing the kernel, there are some problems with that approach as well. Most notably, in some cases packets have to be re-injected back to the kernel when they are destined for ssh or other system services. XDP provides an in-kernel mechanism for packet processing for certain use cases by providing access to the raw packets, so BPF filters can make decisions based on the headers or contents within the packets. XDP programs run in the network driver, which enables them to read an ethernet frame from the Rx ring of the NIC and take actions before any memory is allocated for the packet in the kernel's socket buffers (skb). As a use case, XDP programs can drop packets in the event of denial-of-service attacks at the line rate without overwhelming the kernel's TCP stack or the userspace application. It is important to note that XDP is designed to cooperate with the existing networking subsystem of the kernel, and so developers can selectively use XDP to implement certain features that don't need to leave the kernel space.

XDP programs currently can make the following decisions:

1. XDP_DROP—Instructs the driver to simply drop the packet. It's essentially recycling a page in the Rx ring queue, since this happens at the earliest possible stage of the Rx flow.
2. XDP_TX—Retransmits a packet on the same NIC source. In most scenarios the eBPF program alters the headers or the contents of the packet before retransmitting. This allows for some very interesting use cases, such as load balancing where the load balancing decision is entirely done in the eBPF program. In this scenario, the networking stack or any userspace code doesn't need to participate in the decision making or packet retransmission flow, which allows for throughput close to line rate. One important point to keep in mind is that the XDP infrastructure doesn't have any sort of buffer, because the packets are processed at the driver layer, so when a packet is retransmitted and the TX device is slower, packets might simply get dropped.
3. XDP_PASS—The eBPF program has allowed the packet to move on to the networking stack of the kernel. It's also possible to rewrite the contents of the packets before the packet is passed on.
4. XDP_ABORT—This action is reserved for usage when the program encounters some form of an internal error; it essentially results in the packet getting dropped.

XDP depends on drivers to implement the Rx hook and plug into the eBPF infrastructure. Currently, there can be only one XDP program attached to a driver, but programs can call other programs using the tail calls infrastructure.

Case Study: XDP-Based Firewall

To demonstrate how XDP programs work, we can go through the design of a simple packet filtering service. Services like firewalls are usually divided into a distributed control plane and a data plane. The control plane provides APIs for operators to create filtering rules and introspects the filters to provide telemetry data. The data plane runs on every host in a cluster where packet filtering happens. XDP filters naturally constitute the data plane of such a system.

In general, software using BPF filters are divided into three parts:

1. BPF filter code and maps that are loaded into the kernel
2. Userspace program that loads the filter and provide APIs to update various maps
3. Optional processes like command line tools to access the maps

BPF Maps

The BPF maps form the most essential part of the firewall system. As stated above, they essentially allow the userspace processes to provide the rule set for performing packet filtering and the XDP program to emit telemetry data. We use the following maps in the data plane:

1. LPM trie map—The trie data structure allows doing prefix-based lookups efficiently, and BPF includes an implementation of LPM (longest prefix match) trie. We will use the LPM trie map to store the CIDR blocks of the source and the destination addresses which have to be either blacklisted or whitelisted.
2. Map array—For whitelisted or blacklisted destination ports.
3. Hash maps—Hold counters for packets dropped and passed based on the rule set.

XDP Filters

The BPF program that XDP invokes when a packet is received in the driver contains the logic for parsing incoming packets, reads the maps to look up the rules provided by the userspace process, and makes filtering decisions based on them. The BPF program also updates maps with telemetry data to provide observability into the actions taken.

There would be separate XDP filters for whitelisting and blacklisting flows, so we will have two different XDP filters:

1. The blacklisting filter would parse the ethernet frame and extract the source IP address and the destination port. If the source IP address has a match in the LPM trie, it would simply return the XDP_DROP action. From there on, it would look up the blacklist's array map and return the XDP_DROP action if there is a match. If none of the above checks has a positive outcome, the filter returns the XDP_PASS action, thereby passing on the packet to the kernel's networking stack.
2. The whitelisting filter behaves similarly except that it returns the XDP_PASS action and allows the packet to pass into the kernel only if the lookups within the LPM trie map and the array map have a successful match. In other cases it returns the XDP_DROP action, thereby dropping the packet.

Userspace Program

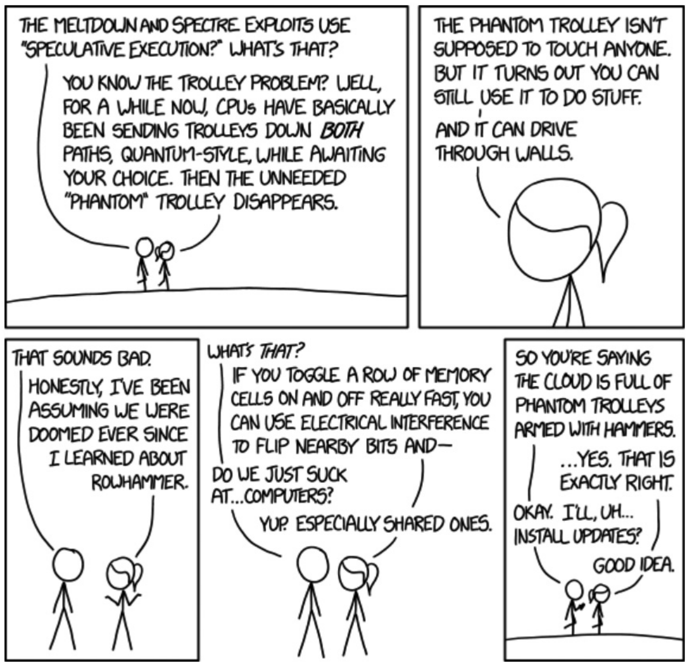
The userspace program has all the necessary infrastructure to interact with the control plane service and also interacts with the BPF infrastructure. It retrieves the rules that need to be enforced, creates the necessary maps, and loads either of the whitelisting or the blacklisting filters based on the rules that need to be enforced. Any updates from the control plane would in turn update the maps containing the rules so that the filters can enforce the new rules. It can also provide APIs for other tools, such as monitoring system APIs that get telemetry data.

In addition to the XDP program and the userspace process that loads it, there could also be additional userspace tools that might interact with the pinned BPF objects. For example, third-party monitoring system tools could implement logic to read the maps and push telemetry data.

Conclusion

eBPF and XDP have been a major step towards achieving programmability in the kernel's data path, which provides safety without compromising on speed. Beyond networking, eBPF has made a significant improvement in the tracing capabilities in the kernel, which has enabled instrumentations in areas that were previously not possible. The future of eBPF in the kernel is strong, and we will see more tools using the power of the BPF infrastructure.

XKCD **xkcd.com**


THE MELTDOWN AND SPECTRE EXPLOITS USE "SPECULATIVE EXECUTION?" WHAT'S THAT?
YOU KNOW THE TROLLEY PROBLEM? WELL, FOR A WHILE NOW, CPUs HAVE BASICALLY BEEN SENDING TROLLEYS DOWN BOTH PATHS, QUANTUM-STYLE, WHILE AWAITING YOUR CHOICE. THEN THE UNNEEDED "PHANTOM" TROLLEY DISAPPEARS.
THE PHANTOM TROLLEY ISN'T SUPPOSED TO TOUCH ANYONE. BUT IT TURNS OUT YOU CAN STILL USE IT TO DO STUFF.
AND IT CAN DRIVE THROUGH WALLS.
THAT SOUNDS BAD.
HONESTLY, I'VE BEEN ASSUMING WE WERE DOOMED EVER SINCE I LEARNED ABOUT ROWHAMMER.
WHAT'S THAT?
IF YOU TOGGLE A ROW OF MEMORY CELLS ON AND OFF REALLY FAST, YOU CAN USE ELECTRICAL INTERFERENCE TO FLIP NEARBY BITS AND—
DO WE JUST SUCK AT...COMPUTERS?
YUP. ESPECIALLY SHARED ONES.
SO YOU'RE SAYING THE CLOUD IS FULL OF PHANTOM TROLLEYS ARMED WITH HAMMERS.
...YES. THAT IS EXACTLY RIGHT.
OKAY. I'LL, UH... INSTALL UPDATES?
GOOD IDEA.

Save the Date!

usenix

LISA18

October 29–31, 2018
Nashville, TN, USA

LISA: Where systems engineering and operations professionals share real-world knowledge about designing, building, and maintaining the critical systems of our interconnected world.

The Call for Participation is now available.
Submissions are due May 24, 2018.

Program Co-Chairs

Rikki Endsley
Opensource.com

Brendan Gregg
Netflix

www.usenix.org/lisa18

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION