**A thorough introduction to eBPF**

**December 2, 2017**

This article was contributed by [Matt Fleming](http://www.codeblueprint.co.uk/)

BPF in the kernel

In his [linux.conf.au 2017 talk [YouTube]](https://www.youtube.com/watch?v=JRFNIKUROPE) on the eBPF in-kernel virtual machine, Brendan Gregg proclaimed that "super powers have finally come to Linux". Getting eBPF to that point has been a long road of evolution and design. While eBPF was originally used for network packet filtering, it turns out that running user-space code inside a sanity-checking virtual machine is a powerful tool for kernel developers and production engineers. Over time, new eBPF users have appeared to take advantage of its performance and convenience. This article explains how eBPF evolved how it works, and how it is used in the kernel.

**The evolution of eBPF**

The original [Berkeley Packet Filter (BPF) [PDF]](http://www.tcpdump.org/papers/bpf-usenix93.pdf) was designed for capturing and filtering network packets that matched specific rules. Filters are implemented as programs to be run on a register-based virtual machine.

The ability to run user-supplied programs inside of the kernel proved to be a useful design decision but other aspects of the original BPF design didn't hold up so well. For one, the design of the virtual machine and its instruction set architecture (ISA) were left behind as modern processors moved to 64-bit registers and invented new instructions required for multiprocessor systems, like the atomic exchange-and-add instruction (XADD). BPF's focus on providing a small number of RISC instructions no longer matched the realities of modern processors.

So, Alexei Starovoitov introduced the [extended BPF (eBPF) design](https://lwn.net/Articles/599755/) to take advantage of advances in modern hardware. The eBPF virtual machine more closely resembles contemporary processors, allowing eBPF instructions to be mapped more closely to the hardware ISA for improved performance. One of the most notable changes was a move to 64-bit registers and an increase in the number of registers from two to ten. Since modern architectures have far more than two registers, this allows parameters to be passed to functions in eBPF virtual machine registers, just like on native hardware. Plus, a new BPF\_CALL instruction made it possible to call in-kernel functions cheaply.

The ease of mapping eBPF to native instructions lends itself to just-in-time compilation, yielding improved performance. The [original patch that added support for eBPF](https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bd4cf0ed331a275e9bf5a49e6d0fd55dffc551b8) in the 3.15 kernel showed that eBPF was up to four times faster on x86-64 than the old classic BPF (cBPF) implementation for some network filter microbenchmarks, and most were 1.5 times faster. Many architectures support the just-in-time (JIT) compiler (x86-64, SPARC, PowerPC, ARM, arm64, MIPS, and s390).

Originally, eBPF was only used internally by the kernel and cBPF programs were translated seamlessly under the hood. But with [commit daedfb22451d](https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=daedfb22451dd02b35c0549566cbb7cc06bdd53b) in 2014, the eBPF virtual machine was [exposed directly to user space](https://lwn.net/Articles/603983/).

**What can you do with eBPF?**

An eBPF program is "attached" to a designated code path in the kernel. When the code path is traversed, any attached eBPF programs are executed. Given its origin, eBPF is especially suited to writing network programs and it's possible to write programs that attach to a network socket to filter traffic, to classify traffic, and to run network classifier actions. It's even possible to modify the settings of an established network socket with an eBPF program. The [XDP project](https://www.iovisor.org/technology/xdp), in particular, uses eBPF to do high-performance packet processing by running eBPF programs at the lowest level of the network stack, immediately after a packet is received.

Another type of filtering performed by the kernel is restricting which system calls a process can use. This is done with [seccomp BPF](https://lwn.net/Articles/656307/).

eBPF is also useful for debugging the kernel and carrying out performance analysis; programs can be attached to tracepoints, kprobes, and perf events. Because eBPF programs can access kernel data structures, developers can write and test new debugging code without having to recompile the kernel. The implications are obvious for busy engineers debugging issues on live, running systems. It's even possible to use eBPF to debug user-space programs by using [Userland Statically Defined Tracepoints](http://blog.memsql.com/bpf-linux-performance/).

The power of eBPF flows from two advantages: it's fast and it's safe. To fully appreciate it, you need to understand how it works.

**The eBPF in-kernel verifier**

There are inherent security and stability risks with allowing user-space code to run inside the kernel. So, a number of checks are performed on every eBPF program before it is loaded. The first test ensures that the eBPF program terminates and does not contain any loops that could cause the kernel to lock up. This is checked by doing a depth-first search of the program's control flow graph (CFG). Unreachable instructions are strictly prohibited; any program that contains unreachable instructions will fail to load.

The second stage is more involved and requires the verifier to simulate the execution of the eBPF program one instruction at a time. The virtual machine state is checked before and after the execution of every instruction to ensure that register and stack state are valid. Out of bounds jumps are prohibited, as is accessing out-of-range data.

The verifier doesn't need to walk every path in the program, it's smart enough to know when the current state of the program is a subset of one it's already checked. Since all previous paths must be valid (otherwise the program would already have failed to load), the current path must also be valid. This allows the verifier to "prune" the current branch and skip its simulation.

The verifier also has a "secure mode" that prohibits pointer arithmetic. Secure mode is enabled whenever a user without the CAP\_SYS\_ADMIN privilege loads an eBPF program. The idea is to make sure that kernel addresses do not leak to unprivileged users and that pointers cannot be written to memory. If secure mode is not enabled, then pointer arithmetic is allowed but only after additional checks are performed. For example, all pointer accesses are checked for type, alignment, and bounds violations.

Registers with uninitialized contents (those that have never been written to) cannot be read; doing so cause the program load to fail. The contents of registers R0-R5 are marked as unreadable across functions calls by storing a special value to catch any reads of an uninitialized register. Similar checks are done for reading variables on the stack and to make sure that no instructions write to the read-only frame-pointer register.

Lastly, the verifier uses the eBPF program type (covered later) to restrict which kernel functions can be called from eBPF programs and which data structures can be accessed. Some program types are allowed to directly access network packet data, for example.

**The bpf() system call**

Programs are loaded using the [bpf() system call](http://man7.org/linux/man-pages/man2/bpf.2.html) with the BPF\_PROG\_LOAD command. The prototype of the system call is:

int bpf(int cmd, union bpf\_attr \*attr, unsigned int size);

The bpf\_attr union allows data to be passed between the kernel and user space; the exact format depends on the cmd argument. The size argument gives the size of the bpf\_attr union object in bytes.

Commands are available for creating and modifying eBPF maps; maps are the generic key/value data structure used for communicating between eBPF programs and the kernel or user space. Additional commands allow attaching eBPF programs to a control-group directory or socket file descriptor, iterating over all maps and programs, and pinning eBPF objects to files so that they're not destroyed when the process that loaded them terminates (the latter is used by the tc classifier/action code so that eBPF programs persist without requiring the loading process to stay alive). The full list of commands can be found in the bpf() man page.

Though there appear to be many different commands, they can be broken down into three categories: commands for working with eBPF programs, working with eBPF maps, or commands for working with both programs and maps (collectively known as objects).

**eBPF program types**

The type of program loaded with BPF\_PROG\_LOAD dictates four things: where the program can be attached, which in-kernel helper functions the verifier will allow to be called, whether network packet data can be accessed directly, and the type of object passed as the first argument to the program. In fact, the program type essentially defines an API. New program types have even been created purely to distinguish between different lists of allowed callable functions (BPF\_PROG\_TYPE\_CGROUP\_SKB versus BPF\_PROG\_TYPE\_SOCKET\_FILTER, for example).

The current set of eBPF program types supported by the kernel is:

* BPF\_PROG\_TYPE\_SOCKET\_FILTER: a network packet filter
* BPF\_PROG\_TYPE\_KPROBE: determine whether a kprobe should fire or not
* BPF\_PROG\_TYPE\_SCHED\_CLS: a network traffic-control classifier
* BPF\_PROG\_TYPE\_SCHED\_ACT: a network traffic-control action
* BPF\_PROG\_TYPE\_TRACEPOINT: determine whether a tracepoint should fire or not
* BPF\_PROG\_TYPE\_XDP: a network packet filter run from the device-driver receive path
* BPF\_PROG\_TYPE\_PERF\_EVENT: determine whether a perf event handler should fire or not
* BPF\_PROG\_TYPE\_CGROUP\_SKB: a network packet filter for control groups
* BPF\_PROG\_TYPE\_CGROUP\_SOCK: a network packet filter for control groups that is allowed to modify socket options
* BPF\_PROG\_TYPE\_LWT\_\*: a network packet filter for lightweight tunnels
* BPF\_PROG\_TYPE\_SOCK\_OPS: a program for setting socket parameters
* BPF\_PROG\_TYPE\_SK\_SKB: a network packet filter for forwarding packets between sockets
* BPF\_PROG\_CGROUP\_DEVICE: determine if a device operation should be permitted or not

As new program types were added, kernel developers discovered a need to add new data structures too.

**eBPF data structures**

The main data structure used by eBPF programs is the eBPF map, a generic data structure that allows data to be passed back and forth within the kernel or between the kernel and user space. As the name "map" implies, data is stored and retrieved using a key.

Maps are created and manipulated using the bpf() system call. When a map is successfully created, a file descriptor associated with that map is returned. Maps are normally destroyed by closing the associated file descriptor. Each map is defined by four values: a type, a maximum number of elements, a value size in bytes, and a key size in bytes. There are different map types and each provides a different behavior and set of tradeoffs:

* BPF\_MAP\_TYPE\_HASH: a hash table
* BPF\_MAP\_TYPE\_ARRAY: an array map, optimized for fast lookup speeds, often used for counters
* BPF\_MAP\_TYPE\_PROG\_ARRAY: an array of file descriptors corresponding to eBPF programs; used to implement jump tables and sub-programs to handle specific packet protocols
* BPF\_MAP\_TYPE\_PERCPU\_ARRAY: a per-CPU array, used to implement histograms of latency
* BPF\_MAP\_TYPE\_PERF\_EVENT\_ARRAY: stores pointers to struct perf\_event, used to read and store perf event counters
* BPF\_MAP\_TYPE\_CGROUP\_ARRAY: stores pointers to control groups
* BPF\_MAP\_TYPE\_PERCPU\_HASH: a per-CPU hash table
* BPF\_MAP\_TYPE\_LRU\_HASH: a hash table that only retains the most recently used items
* BPF\_MAP\_TYPE\_LRU\_PERCPU\_HASH: a per-CPU hash table that only retains the most recently used items
* BPF\_MAP\_TYPE\_LPM\_TRIE: a longest-prefix match trie, good for matching IP addresses to a range
* BPF\_MAP\_TYPE\_STACK\_TRACE: stores stack traces
* BPF\_MAP\_TYPE\_ARRAY\_OF\_MAPS: a map-in-map data structure
* BPF\_MAP\_TYPE\_HASH\_OF\_MAPS: a map-in-map data structure
* BPF\_MAP\_TYPE\_DEVICE\_MAP: for storing and looking up network device references
* BPF\_MAP\_TYPE\_SOCKET\_MAP: stores and looks up sockets and allows socket redirection with BPF helper functions

All maps can be accessed from eBPF or user-space programs using the bpf\_map\_lookup\_elem() and bpf\_map\_update\_elem() functions. Some map types, such as socket maps, work with additional eBPF helper functions that perform special tasks.

**How to write an eBPF program**

Historically, it was necessary to write eBPF assembly by hand and use the kernel's bpf\_asm assembler to generate BPF bytecode. Fortunately, the LLVM Clang compiler has grown support for an eBPF backend that compiles C into bytecode. Object files containing this bytecode can then be directly loaded with the bpf() system call and BPF\_PROG\_LOAD command.

You can write your own eBPF program in C by compiling with Clang using the -march=bpf parameter. There are plenty of eBPF program examples in the kernel's [samples/bpf/](http://elixir.free-electrons.com/linux/v4.14.2/source/samples/bpf) directory; the majority have a "\_kern.c" suffix in their file name. The object file (eBPF bytecode) emitted by Clang needs to be loaded by a program that runs natively on your machine (these samples usually have "\_user.c" in their filename). To make it easier to write eBPF programs, the kernel provides the libbpf library, which includes helper functions for loading programs and creating and manipulating eBPF objects. For example, the high-level flow of an eBPF program and user program using libbpf might go something like:

* Read the eBPF bytecode into a buffer in your user application and pass it to bpf\_load\_program().
* The eBPF program, when run by the kernel, will call bpf\_map\_lookup\_elem() to find an element in a map and store a new value in it.
* The user application calls bpf\_map\_lookup\_elem() to read out the value stored by the eBPF program in the kernel.

However, all of the sample code suffers from one major drawback: you need to compile your eBPF program from within the kernel source tree. Luckily, the BCC project was created to solve this problem. It includes a complete toolchain for writing eBPF programs and loading them without linking against the kernel source tree.

BCC is covered in the next article in this series; the full set is:

* [An introduction to the BPF Compiler Collection](https://lwn.net/Articles/742082/)
* [Some advanced BCC topics](https://lwn.net/Articles/747640/)
* [User-space tracepoints with BPF](https://lwn.net/Articles/753601/)