Summary of the paper “eBPF-From a Programmers Perspective”

1: - eBPF allows execution of the programs in the kernel without requiring recompilation and system restart and offer a simpler and safer interface than module programming. 2: - ***Berkeley Packet Filter***or *BPF* is a network packet filter which is a network security mechanism for controlling what flows from and to a network by inspecting packets as they pass through the filter. 3: - Some good features of eBPF 3.1: - Static verifier that ensures that a program cannot cause a kernel crash and that it will  
always terminate. 3.2: - Execution point definition**.** Before the kernel can run eBPF programs, it must know where to attach it. They are defined by the eBPF program types. 3.3: -Maps The eBPF architecture also contains maps, which are bidirectional data structures allowing eBPF programs to asynchronously share data with user space. 3.3.1: - Maps are created by invoking the ***bpf* syscall** with the BPF\_MAP\_CREATE argument [4]. One can also make use the SEC attribute discussed earlier to automatically create. 3.3.2: - An interesting property of eBPF maps is the in-kernel aggregation. This significantly decreases the overhead compared to systems that transfer all samples to user-space for processing. 3.3.3: - Different map types discussed 3.3.3.1: - Array type 3.3.3.2: - hash type 3.3.4: -The user-space program can interact with maps by using the three methods 3.3.4.1: -bpf\_map\_lookup\_elem () 3.3.4.2: -bpf\_map\_update\_elem () 3.3.4.3: -bpf\_map\_delete\_elem ()

4: - The *SEC* macro in the code of libbpf is used to tell the compiler to place the bytecode in a specified ELF section. 5: - Focus on C and compilation using *clang* and *llc* because Clang have had a longer history with eBPF. 6: - Libbpf offers an interface in C to work with eBPF. It offers functions to build a struct *bpf\_object* by reading the bytecode of a program and the associated metadata. 7: - Networking eBPF programs are used to read, modify, retransmit, redirect or drop network packets. The actions that can be performed on the packet (cloning, retransmission, redirection, ...) and the amount of data. Different types exist in these programs. 7.1: - ***Socket Filter Programs****.* This type enables an eBPF program to attach to sockets and  
read packets going through the socket. It also allows truncation and dropping of packets.

7.2: - *XDP Programs.* The eBPF XDP type enables eBPF programs to inspect incoming network packets early in the network stack. XDP programs can return

7.2.1: - *XDP\_PASS* to allow it to continue to the next subsystem. 7.2.2: - *XDP\_DROP* to drop it. 7.2.3: - *XDP\_TX* to forward it back to the NIC that originally received it.   
7.2.4: - *XDP\_REDIRECT* to send the packet through a different NIC and possibly bypass the normal network stack.

8: -***Tracing*** eBPF programs are used to debug or trace performance of either the kernel or user-space applications.

8.1: -*Tracepoint Programs.* This type enables eBPF programs to attach to the tracepoints handler provided by the kernel. Tracepoints are static marks in the kernel that can be used for tracing and  
debugging purposes and defined in the /sys/kernel/debug/tracing/events directory. 8.2: -*Raw Tracepoint Programs* This type works like the *Tracepoint* type, but can  
access the tracepoint more directly. 8.3: *-Kprobe Programs.* The eBPF Kprobe type enables eBPF programs to dynamically attach to any function in the kernel. ***The most important difference between kprobes and tracepoints is that tracepoints are statically defined in the kernel while kprobes can be placed in any named function in the kernel.*** 8.4: - *Perf Event Programs.* The eBPF Perf Event type allows eBPF programs to attach to the kernel’s internal *Perf* profiler. Perf emits performance data events for hardware and software. 9: - *Hardening step* After verification, the kernel memory holding the eBPF program is made read-only to protect from malicious manipulation. 10: - *Risky Operations:* When writing a user-space C program, invalid memory accesses are caught as segmentation faults. In eBPF programs, invalid memory accesses must not happen in any circumstances. the program must use the bpf\_probe\_read function. This function will verify that the pointer is valid and copy the desired memory space before continuing the program execution 11: - Some Practical use cases of eBPF 11.1:- DDos Firewall 11.2: - Cilium 11.3: - ExtFUSE 11.4: -Katran 11.5: -Bcc 11.6: - bpftrace

12: -Some of the NEXT STEPS an eBPF developer can take.

12.1: - *High-level inspection of eBPF objects.* Use tools such as bpftool to view and debug eBPF programs and maps.  
12.2: -*Low-level inspection of eBPF programs.* Part of understanding and debugging the behavior of eBPF programs is dumping and reading the bytecode by using tools such as *llvmobjdump.*  
12.3: - *Evaluate performance of eBPF programs.* It is valuable for an eBPF developer to understand the overhead of eBPF programs. In Linux, one can enable a flag that will collect performance metrics about loaded eBPF programs.  
12.4: - *CO-RE. Compile Once - Run Everywhere.* Allows eBPF developers to compile code on a system to be run on any other system with varying operating systems and kernel versions.