LF live

MENTORSHIP SERIES

LF Live MENTORSHIP SERIES

Kernel Tracing using eBPF

Vandana Salve, Linux Kernel Engineer

Introduction and Agenda

- Basics of eBPF architecture
- Dynamic and static instrumentation
- Kernel instrumentation
- Using BCC and Bpftrace tools for kernel tracing

Scope and Expectations

- High level introduction of eBPF architecture
- Pointers to useful information and projects
- Using eBPF project tools for kernel code tracing

Background

- Originally created to speed up the filtering of network packets, known as BPF – Berkeley Packet Filter.
- In 2014, Alexei Starovoitov redesigned the language of BPF into extended BPF functionality
  - Turning BPF into general-purpose execution engine, that can be used for variety of things
  - Beyond packet filtering can be used for profiling, debugging, security policies etc.
- The eBPF included in Linux kernel in v3.18 in 2014 with features
  - Bpf() system call to load the BPF programs
  - Support for different kinds of BPF programs, not just for packet filtering.
    - Tracing, profiling and security
- BPF evolved to what we call as “extended BPF: or “eBPF”

eBPF Basics

- eBPF is a revolutionary kernel technology that allows developers to write custom code that can be loaded into the kernel dynamically,
  - Changing the way the kernel behaves
  - Provides the flexibility to build bespoke tools or customized policies
- eBPF-based tools can observe any event across the kernel and all applications on hw/virtual machine or containerised
- An efficient technology composed of
  - Instruction set, storage objects and helper functions
- It can be consider as virtual machine, running code in an isolated environment

Why eBPF is important? Difference between kernel and user space in Linux

- The applications run in unprivileged layer called user space, can’t access h/w directly
    - Uses “system calls” to request kernel to act on its behalf
    - H/W access can involve
        - Reading/writing files, sending/receiving packets
        - Just accessing memory or several other activities
- The kernel runs in privilege layer, a layer between your application and the hardware
    - Abstract the hardware or virtual hardware
    - Wide set of subsystems and layers, handling the resources
    - Each subsystem typically allows for some level of configuration to account for different needs of users
- If a desired behavior cannot be configured, a kernel change is required, historically leaving 2 options
    - Adding change in kernel – native support
    - Adding change dynamically – kernel modules

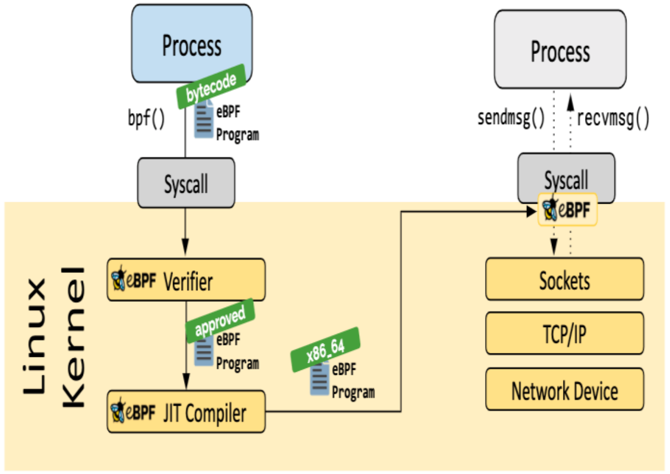
Native change Vs Kernel module Vs BPF

- Native support
  - Make the kernel change
  - Convince the kernel community, why the change is necessary
  - Wait several years to get the change reviewed/accepted and available in upstream kernel
- Kernel module
  - Make the kernel change as a kernel module
  - Need maintenance/fix as and when kernel APIs changes
  - Risk corrupting your kernel due to lack of security boundaries
- eBPF
  - Allows reprogramming the behavior of kernel without requiring changes to kernel sources or loading kernel module

LF Live MENTORSHIP SERIES

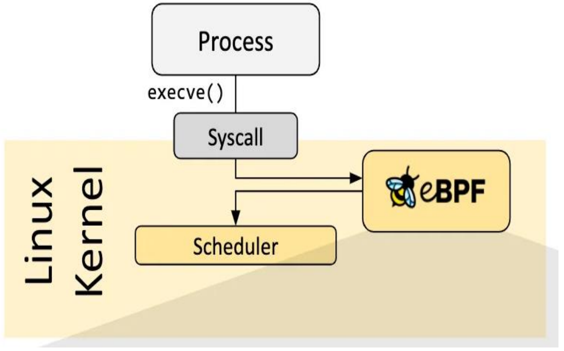
BPF architecture

- eBPF programs are loaded and executed in kernel
- eBPF verifier
  - Ensures the BPF programs are safe and crash-free
  - Rejects if it is deemed to be unsafe
    - Not allowing looping or backward branches
- JIT compiler
  - Translates the generic byte-code of the program into machine specific instruction set
- eBPF MAPS
  - Provides ability to share the collected information between the kernel and user space
  - Supports various types of maps
    - Hash tables, arrays, ring buffers, stack trace etc.



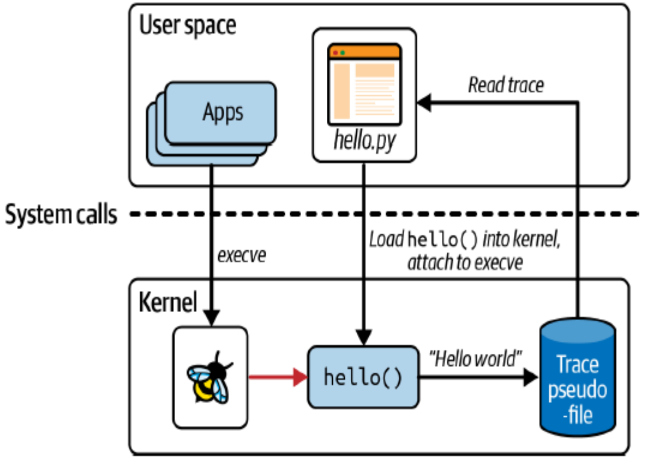
Dynamic loading of eBPF program

- eBPF programs can be loaded into and removed from the kernel dynamically.
- Once they are attached to the event, triggered regardless of the cause of the event
- Pre-defined hooks
  - System calls, functions @ Entry/Exit
  - Kernel trace points
  - Network events and several other places
- Example
  - Attach the BPF program at the entry of ‘open(2)’ system call
    - It will be triggered whenever any process tries to open the file
  - Attach the BPF program at the entry of ‘execve(2)’ system call
    - It will be triggered whenever a new binary application is executed
- This leads to one of the greatest strength of observability or security tooling that uses eBPF – gives instant visibility over everything happening on the machine



Lets get started with Greetings : Hello World program

- Python code example : hello.py application, consists 2 parts
- eBPF Program, that will run in the kernel
  - Written in C, ***hello*** function
  - It just calls a helper function ***bpf_trace_print()***
  - Entire eBPF program defined as **string *program*** in python code, which needs to be passed, when creating BPF object
- Some user space code, i.e. python program that loads the eBPF program
  - BCC takes care of compiling the C-code before executing it
  - Compiles the C code and loads it into the kernel
  - Attaches it to the ***execve*** syscall kprobe
  - Reads out the trace that it generates



Running the program : Hello World

- Run this program and depending on what's happening on the machine, get to see the tracing prints generated as other processes are created by **execve** system call execution
- Trace output shows not only ***Hello World*** string but also additional context information about the event that triggered eBPF program such as Process ID
- For trace messages, the contextual information is added as part of the kernel tracing infrastructure
- Python code reads the trace out from pseudo file ***/sys/kernel/debug/tracing/trace_pipe***

BPF Maps

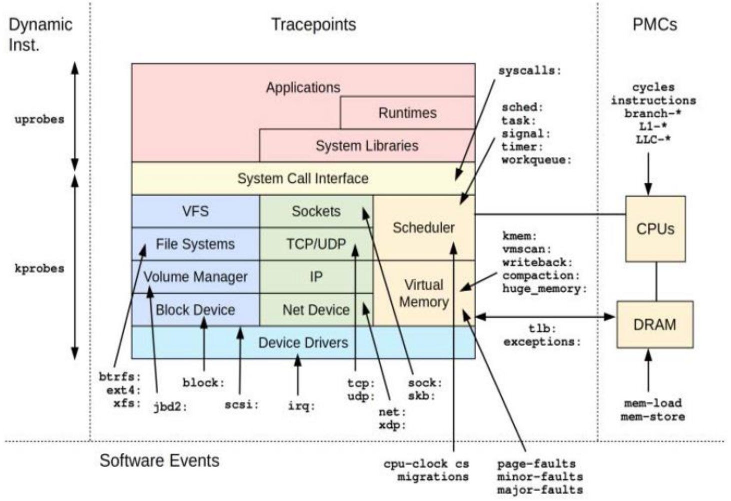
- For passing structure information, **trace_pipe** might not be sufficient, in that case use BPF Maps
- BPF map is a data structure accessed from eBPF programs and from user space, key-value stores
  - Various types defined in *uapi/linux/bpf.h*
  - Hash tables, ring buffers, arrays etc.
  - Some optimized for particular types of operations : FIFO queues, FILO stacks, LRU data stores
  - Some are information specific : sockmaps, devmaps: socket and network device information

Tracing

- Tracing can be used for debugging and troubleshooting
- Traditional tools
  - Ftrace
  - Perf
  - Strace and ltrace
  - Sar etc
- These tools can provide useful starting point for analysis
  - Can be used for more exploration with BPF tracing tools

Tracing with BPF

- Tracing gives visibility across full software stack and allows to collect data for profiling and debugging/troubleshooting
  - System call interface, file systems, virtual memory, scheduler, networking etc.
- BPF tracing supports multiple sources of events to provide visibility of the entire software stack
- Lets looks at the 4 instrumentation ways to look deeper into the system
  - Dynamic instrumentation
    - Kernel instrumentation with kprobes
    - User space instrumentation with uprobes
  - Static instrumentation
    - Kernel instrumentation with tracepoints
    - User space instrumentation with USDT



Dynamic instrumentation	**Static instrumentation**
Ability to insert instrumentation points into live software, in production	The instrumentation points are coded into the software and maintained by the developers
Dynamic instrumentation for kernel was added in the form of **kprobes** in v2.6.9	Static instrumentation for kernel was added in the form of **tracepoints** in v2.6.32
Dynamic instrumentation for user-level function was added in the form **uprobes**	For user space applications was added in the form of **USDT-user statically defined tracepoints**
Use to instrument start/end of kernel and application functions	Use to instrument/add static trace points in kernel/user application functions
kernel probes don't have a stable application binary interface (ABI)	As they are static, the ABI for tracepoints are stable

Kprobes and kretprobes

- Kprobes allows to set hooks in almost any instruction in the kernel code
  - Commonly eBPF programs attached to using
    - **kprobes** to the entry of functions (or at specific offset after the entry)
      - BCC : attach_probe()
      - Bpftrace: kprobe:function_name[+offset]
      - BCC: attach_kretprobe()
      - **bpftrace -e 'kprobe:do_sys_open { printf("%s opening: %s\n", comm, str(arg1)); }**
    - **Kretprobes** to exit of function
      - BCC: attach_kretprobe()
      - Bpftrace: kretprobe:function_name
      - **bpftrace -e 'kretprobe:vfs_read { @bytes = hist(retval);}**
- For debugging & performance, developers can write kernel modules that attached functions to kprobes

Tracepoints – Static instrumentation

- Inserted into the kernel code at logical places by the developers – static markers
- List of tracepoints : **/sys/kernel/debug/tracing/available_events**
- Available set of tracepoints : **cat /sys/kernel/debug/tracing/available_events**

```
syscalls:sys_exit_clone
syscalls:sys_enter_clone
syscalls:sys_exit_vfork
syscalls:sys_enter_vfork
syscalls:sys_exit_fork
syscalls:sys_enter_fork
```

- Trace a process ‘sshd” performing read of number of bytes, using arguments
  - # **bpftrace -e 'tracepoint:syscalls:sys_enter_read /comm == "sshd"/ {printf("%s doing reads of : %d byte buffer\n", comm, args->count);}’**

BPF development toolchain

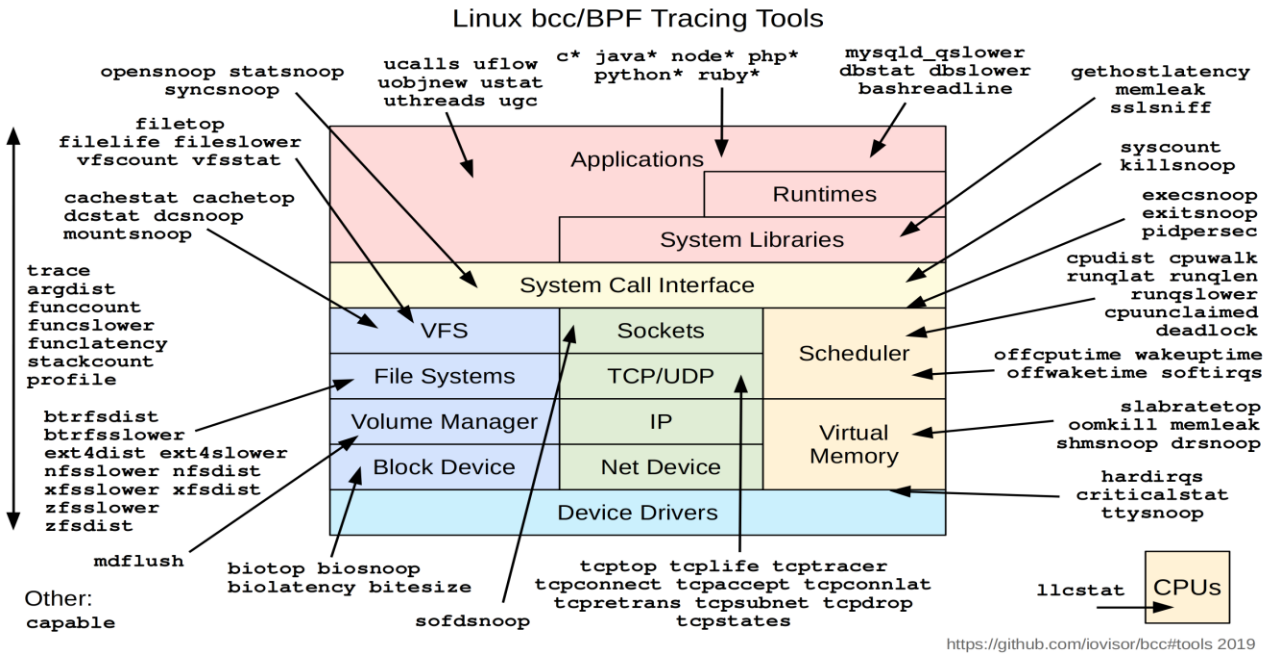
- In lot of scenarios, eBPF is used indirectly via projects such as **BCC,** or **BPFTrace** or **Cilium** project
    - This provides abstraction on top of eBPF and do not require to write the eBPF programs directly
- Front ends for BPF tracing
    - **BCC** (BPF compiler collection)
    - **BPFTrace** (BPF Trace)
- BPF programs can be written use
    - Psuedo C, LUA, Go or Python language
    - Helper library : libbpf C/C++ library
- BPF programs are complied to byte code using the compilers
    - LLVM
    - clang
- If you looking for tools to run, start with BCC tools and then bpftrace tool
- If you want to write programs, start with bpftrace and then BCC programs

BPF development toolchain

- BCC (BPF compiler collection)
  - Framework that enables users to write a python programs and eBPF programs embedded inside them
  - Running the python program will generate the eBPF bytecode and load it into the kernel
  - Useful for complex tools and daemon applications

- bpftrace
  - bpftrace is a high-level tracing language for Linux eBPF
  - Available since kernel v4.x
  - Uses
    - LLVM as a backend to compile scripts into eBPF bytecode and
    - BCC for interacting with the Linux eBPF subsystem
  - The bpftrace language is inspired by “awk” and “C”
  - Useful for one-liners and short scripts

BCC performance tools



Linux bcc/BPF Tracing Tools
opensnoop statsnoop syncsnoop
ucalls uflow uobjnew ustat uthreads ugc
c* java* node* php* python* ruby*
mysqld_qslower dbstat dbslower bashreadline
gethostlatency memleak sslsniff
filetop filelife fileslower vfscount vfsstat
cachestat cachetop dcstat dcsnoop mountsnoop
syscount killsnoop
execsnoop exitsnoop pidpersec
cpudist cpuwalk runqlat runqlen runqslower cpuunclaimed deadlock
trace argdist funccount funcslower funclatency stackcount profile
offcputime wakeuptime offwaketime softirqs
btrfsdist btrfsslower ext4dist ext4slower nfsslower nfsdist xfsslower xfsdist zfsslower zfsdist
slabratetop oomkill memleak shmsnoop drsnoop
hardirqs criticalstat ttysnoop
mdflush
biotop biosnoop biolatency bitesize
sofdsnoop
tcptop tcplife tcptracer tcpconnect tcpaccept tcpconnlat tcpretrans tcpsubnet tcpdrop tcpstates
llcstat
CPUs
Other: capable
Applications
Runtimes
System Libraries
System Call Interface
VFS
Sockets
Scheduler
File Systems
TCP/UDP
Volume Manager
IP
Virtual Memory
Block Device
Net Device
Device Drivers
https://github.com/iovisor/bcc#tools 2019

Some BCC tools exploration – System call interface

- opensnoop - Shows which processes are attempting to open which files
    - Trouble shooting application that are failing, any misconfiguration?
- statsnoop - Traces different **stat(2)** system calls, shows which processes are attempting to read information about which files
- Execsnoop - Traces new processes via **exec(2)** system calls

- Filetop : traces file read/writes and prints summary every interval
    - Traces __vfs_read and __vfs_write() function using dynamic tracing
- Vfsccount: counts VFS calls: useful for general workload characterization
    - Traces all kernel functions beginning with “vfs_” using dynamic tracing
- Vfsstat: Traces some common VFS calls
    - Traces some common VFS functions

Some BCC tools exploration - Scheduler

- Cpudist : Measure the time the “task” spends on the CPU before descheduled and shows time in histogram
  - Task spending short time on the CPU
    - Indication of excessive context switches and poor workload distribution
    - Possibility of contention for shared resource (such as mutex to become available)
  - Task spending off-CPU time before it is scheduled again
    - Identifying long blocking and I/O operations
  - Uses in-kernel eBPF maps for storing timestamp and histograms
    - May incur overhead for some work loads as its traces scheduler tracepoints

Some BCC tools exploration – Virtual memory

- Memleak
  - Traces and matches the memory allocation/de-allocation requests
  - Collect call stack for each allocation
  - Prints summary of which call stacks performed allocations that weren't freed
  - Tracing a specific process
    - List of allocations from functions from libc such as malloc, calloc, posix_memalign, valloc, free etc.
  - Tracing all processes
    - List of allocations such kmalloc, kfree, kmem_cache_alloc, kmem_cache_free and page allocations APIs

LF Live MENTORSHIP SERIES

bpftrace performance tools

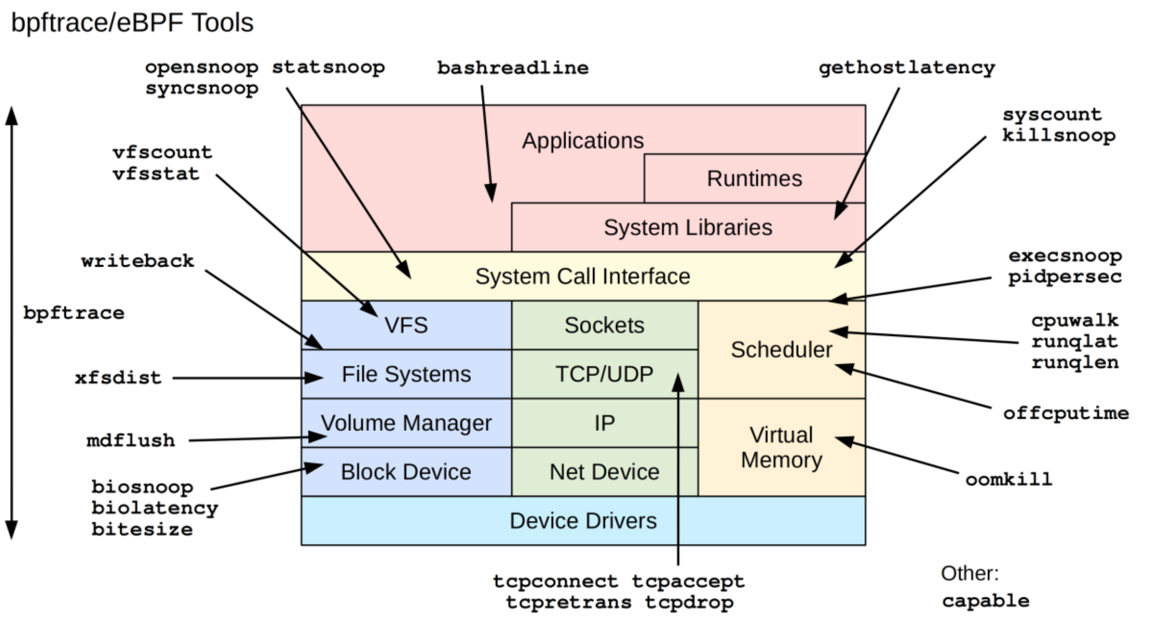


Diagram by Brendan Gregg, early 2019. https://github.com/iovisor/bpftrace

Some bpftrace one-liner examples

- Trace which files are been opened
  - # **bpftrace -e 'kprobe:do_sys_open { printf("%s opening: %s\n", comm, str(arg1)); }'**
- Trace a process ‘sshd’ performing read of number of bytes
  - # **bpftrace -e 'tracepoint:syscalls:sys_enter_read /comm == "sshd"/ {printf("%s doing reads of : %d byte buffer\n", comm, args->count);}'**
- Get system call count by process
  - # **bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'**
- Get Process level events
  - # **bpftrace -e 'tracepoint:sched:sched* { @[probe] = count(); } interval:s:5 { exit(); }'**
  - **Sched:** probe category for high-level scheduler and process events such as **fork, exec** and **context switches**
  - **Probe:** name of the probe
  - **interval:** probe that fires every 5 secs
  - **Exit()**: exit bpftrace

BPF tracing for Network traffic

- Lets look at network traffic on the system for TCP connection
- Generate traffic using curl command
  - $ curl google.com
- Trace the TCP Connection with **tcpconnect**
- It traces active TCP connections via connect() system call, by tracing the kernel
  - **tcp_v4_connect()** and **tcp_v6_connect()** using dynamic tracing

- Example :
  - /usr/sbin/tcpconnect.bt
  - /usr/sbin/tcpconnect-bpfcc

BPF tracing for Network traffic – ICMP and funccount tool

- Lets look at funccount tool - count functions and tracepoints matching the pattern
- Generate traffic using curl command
  - This tool tells us quicky which functions are been called matching the pattern
  - *icmp* functions
  - This uses dynamic tracing and in-kernel maps to count function calls
  - Ftrace can do this but BPF you can add more logic
- # **funccount-bpfcc -i 1 *icmp***
- Lets do some exploration of kernel functions using **trace** tool
  - **icmp_out_count() and icmp_rcv()**
- **Trace** probes functions you specify and display trace messages if particular condition is matched
  - Control the message format to display the function arguments and return values
  - # **trace-bpfcc -t 'icmp_out_count "type: %d" arg2**

BPF tracing for Network traffic – ICMP and trace tool

- Lets do some exploration of kernel functions using **trace** tool
  - **icmp_out_count() and icmp_rcv()**
- **Trace** probes functions you specify and display trace messages if particular condition is matched
  - Control the message format to display the function arguments and return values
  - # **trace-bpfcc -t 'icmp_out_count "type: %d" arg2**
- **void icmp_out_count(struct net *net, unsigned char type)**
  - **Type – icmp message type – Echo request**
  - **#define ICMP_ECHO 8**
  - **Defined in uapi/linux/icmp.h**
- # **trace-bpfcc -I net/net_namespace.h 'icmp_out_count(struct net *net, unsigned char type) "ifindex: %d, type: %d", net->ifindex, type'**

Terms	Descriptions
BPF	Berkeley Packet Filter: a kernel technology originally developed for optimizing the processing of packet filters (eg, tcpdump expressions)
eBPF	Enhanced BPF: a kernel technology that extends BPF so that it can execute more generic programs on any events, such as the bpftrace programs listed below. It makes use of the BPF sandboxed virtual machine environment. Also note that eBPF is often just referred to as BPF.
probe	An instrumentation point in software or hardware, that generates events that can execute bpftrace programs.
static tracing	Hard-coded instrumentation points in code. Since these are fixed, they may be provided as part of a stable API, and documented.
dynamic tracing	Also known as dynamic instrumentation, this is a technology that can instrument any software event, such as function calls and returns, by live modification of instruction text. Target software usually does not need special capabilities to support dynamic tracing, other than a symbol table that bpftrace can read. Since this instruments all software text, it is not considered a stable API, and the target functions may not be documented outside of their source code.
tracepoints	A Linux kernel technology for providing static tracing.
kprobes	A Linux kernel technology for providing dynamic tracing of kernel functions.
uprobes	A Linux kernel technology for providing dynamic tracing of user-level functions.
USDT	User Statically-Defined Tracing: static tracing points for user-level software. Some applications support USDT.
BPF map	A BPF memory object, which is used by bpftrace to create many higher-level objects.

Thanks

- The BCC developers
- Brendan Gregg
- Liz Rice
- eBPF Foundation

LF live MENTORSHIP SERIES

Thank you for joining us today!

We hope it will be helpful in your journey to learning more about effective and productive participation in open source projects. We will leave you with a few additional resources for your continued learning:

- The LF Mentoring Program is designed to help new developers with necessary skills and resources to experiment, learn and contribute effectively to open source communities.
- Outreachy remote internships program supports diversity in open source and free software
- Linux Foundation Training offers a wide range of free courses, webinars, tutorials and publications to help you explore the open source technology landscape.
- Linux Foundation Events also provide educational content across a range of skill levels and topics, as well as the chance to meet others in the community, to collaborate, exchange ideas, expand job opportunities and more. You can find all events at events.linuxfoundation.org.