eBPF und seine Anwendung

Übersicht

- BPF – Historie und Hintergrund
- eBPF – Was ist hat sich geändert
- Anwendungen – Was macht man damit

BPF – Historie I

The BSD Packet Filter:
A New Architecture for User-level Packet Capture*

Steven McCanne† and Van Jacobson†
Lawrence Berkeley Laboratory
One Cyclotron Road
Berkeley, CA 94720
mccanne@ee.lbl.gov, van@ee.lbl.gov

December 19, 1992

Zu finden unter: https://www.tcpdump.org/papers/bpf-usenix93.pdf

BPF – Historie II

Ein ‘virtueller Prozessor’ als ein kernel modul um Pakete zu filtern.

+ Einfache Assembler Struktur
+ Virtueller Adressbereich
  Buffer + Scratch
+ Keine anderen Schnittstellen
=> Sicher im Kernel

3.3 The BPF Pseudo-Machine

The BPF machine abstraction consists of an accumulator, an index register (x), a scratch memory store, and an implicit program counter. The operations on these elements can be categorized into the following groups:

1. LOAD INSTRUCTIONS copy a value into the accumulator or index register. The source can be an immediate value, packet data at a fixed offset, packet data at a variable offset, the packet length, or the scratch memory store.

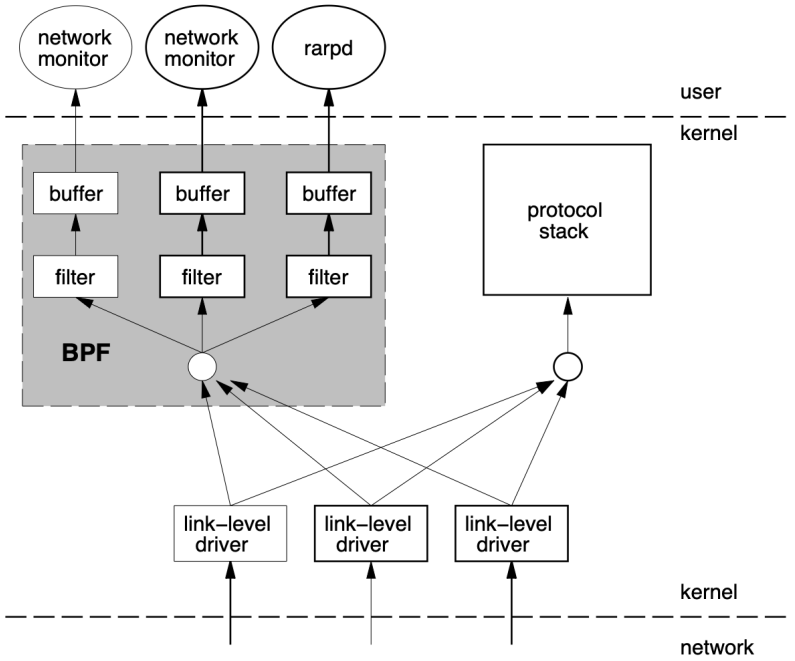


Figure 1: BPF Overview

BPF – Historie III

Weit verbreitet, Teil von tcpdump und libpcap und vielen anderen Nischen.

```
/root: tcpdump -d host 10.27.0.1
(000) ld       [0]
(001) jeq      #0x2000000       jt 2jf 7
(002) ld       [24]
(003) jeq      #0xa1b0001       jt 6jf 4
(004) ld       [28]
(005) jeq      #0xa1b0001       jt 6jf 7
(006) ret      #65535
(007) ret      #0
```

Probleme:

- Kein ‘state’ zwischen Aufrufen
- Keine Schnittstelle zum Rest vom Netzwerkstack
- … und viel anderes …

Aber die Idee ‘Nutzer-Code im Kernel’ hat sich gehalten …

eBPF – Evolution

Ein ‘CPU upgrade’ in eBPF:

+ 64bit Architektur

+ Mehr Register

+ Getuned für JIT (just in time compilation) und kernel ABI,
  insbesondere bei x86_64 und aarch64

Ausserdem Neu:

+ Viele neue Stellen ‘hooks’, an denen eBPF eingehängt werden kann
+ ein ‘validator’ beim laden des Codes in den Kernel
+ Schnittstellen innerhalb des kernels
+ Schnittstellen in den User Space

eBPF – Events

eBPF Programme werden vom Kernel ausgeführt wenn bestimmte Ereignisse (Events) eintreten.

Der kernel bietet VIELE solcher Events

+ Netzwerk (auf verschiedenen Ebenen)

+ Syscall (entry & exit)

+ Storage

+ Trace Points

+ …

```
BPF_PROG_TYPE_UNSPEC,
BPF_PROG_TYPE_SOCKET_FILTER,
BPF_PROG_TYPE_KPROBE,
BPF_PROG_TYPE_SCHED_CLS,
BPF_PROG_TYPE_SCHED_ACT,
BPF_PROG_TYPE_TRACEPOINT,
BPF_PROG_TYPE_XDP,
BPF_PROG_TYPE_PERF_EVENT,
BPF_PROG_TYPE_CGROUP_SKB,
BPF_PROG_TYPE_CGROUP_SOCK,
BPF_PROG_TYPE_LWT_IN,
BPF_PROG_TYPE_LWT_OUT,
BPF_PROG_TYPE_LWT_XMIT,
BPF_PROG_TYPE_SOCK_OPS,
```

Und noch 20 mehr

eBPF – Maps & Helper

Die eBPF runtime stellt ausserdem Funktionen und Datenstrukturen bereit damit der Code mit dem Rest des Kernels und Userspace interagieren kann.

Maps:

HashMap, Array, Histogram, StackTrace, RingBuffer und mehr

Funktionen:

Zeitstempel, UID/GID, RND, kernel/userspace read

eBPF – Code

eBPF Programme werden kompiliert, üblicherweise aus eingeschränktem C code.

By design the eBPF VM and its programs are intentionally not Turing complete:
- no loops are allowed (there is work-in-progress to support bounded loops)
  so each eBPF program is guaranteed to finish and not hang,
- all memory access is bounded and type-checked […]
- there are no null dereferences,
- a program must have at most BPF_MAXINSNS instructions (default 4096),
- the "main" function takes a single argument (the context) and so on.

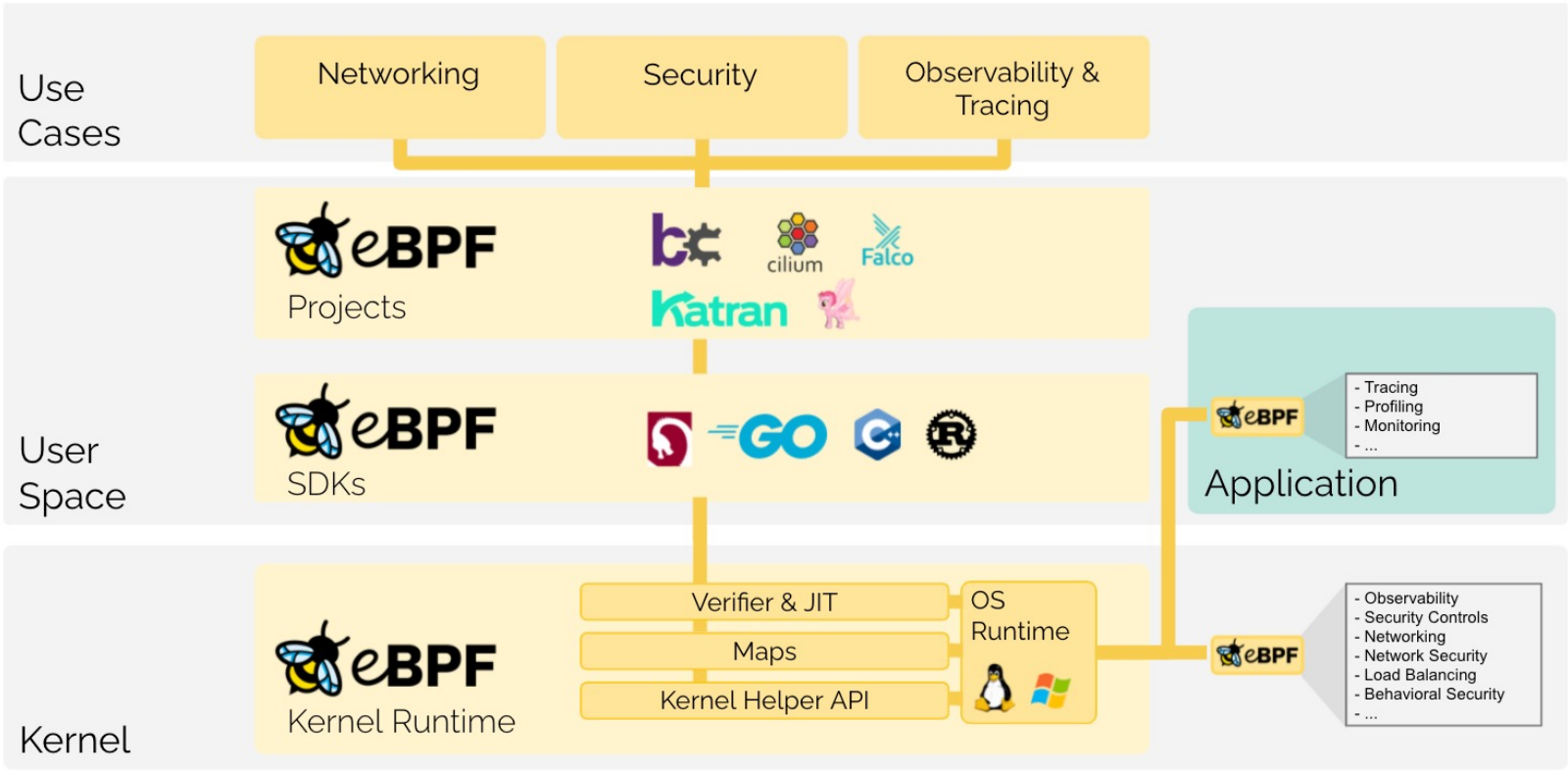
Von: Collabora

Der wichtige Punkt: eBPF erlaubt ‘fast vollwertige’ Programme

Anwendungen

"eBPF does to Linux what JavaScript does to HTML"

Brendan Gregg



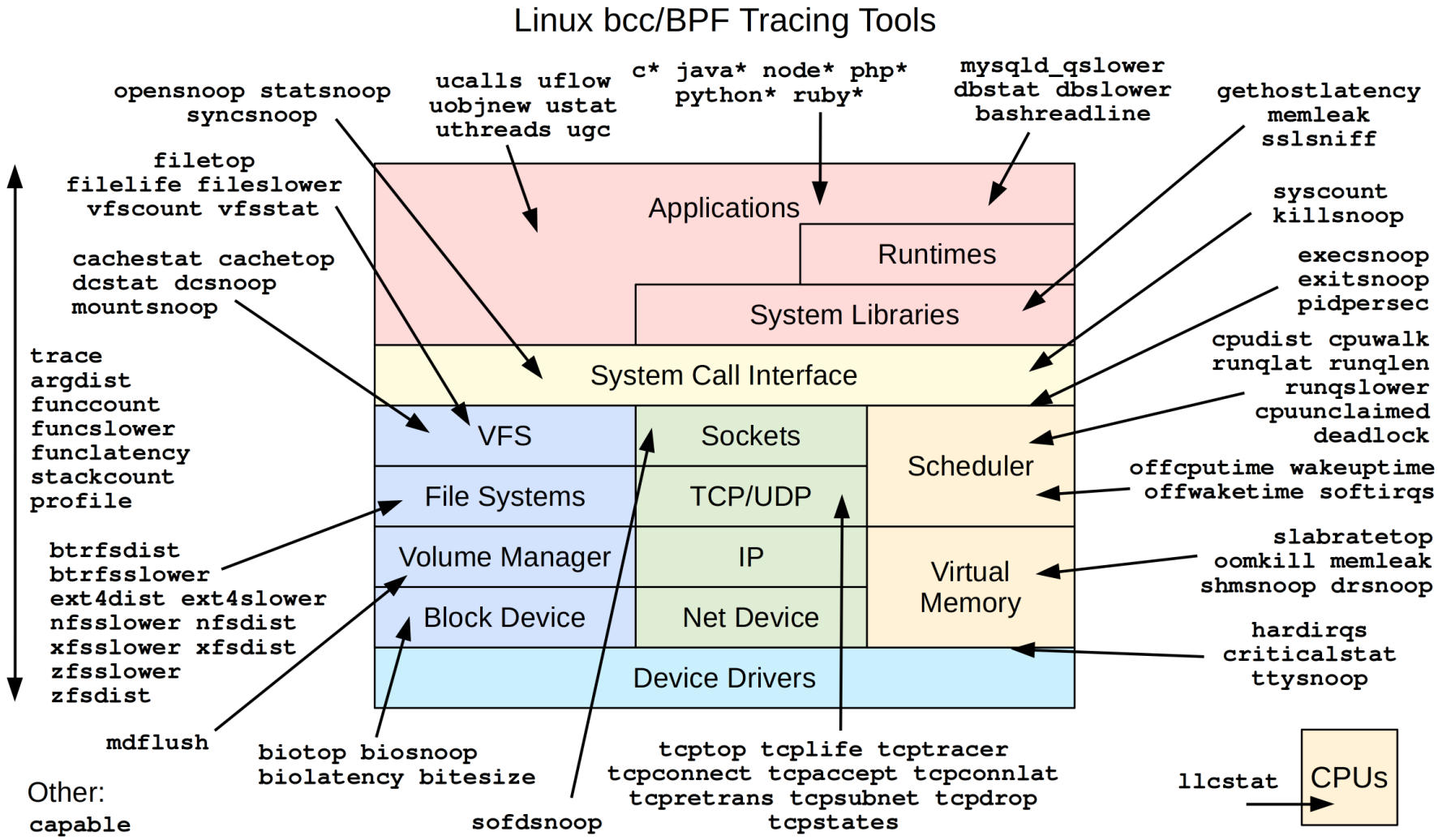
Anwendungen

Beispiel bpftrace - gethostlatency

```
29  uprobe:libc:getaddrinfo,
30  uprobe:libc:gethostbyname,
31  uprobe:libc:gethostbyname2
32  {
33          @start[tid] = nsecs;
34          @name[tid] = arg0;
35  }
36
37  uretprobe:libc:getaddrinfo,
38  uretprobe:libc:gethostbyname,
39  uretprobe:libc:gethostbyname2
40  /@start[tid]/
41  {
42          $latms = (nsecs - @start[tid]) / 1e6;
43          time("%H:%M:%S  ");
44          printf("%-6d %-16s %6d %s\n", pid, comm, $latms, str(@name[tid]));
45          delete(@start[tid]);
46          delete(@name[tid]);
47  }
```

```
# ./gethostlatency.bt
Attaching 7 probes...
Tracing getaddr/gethost calls... Hit Ctrl-C to end.
TIME      PID    COMM               LATms HOST
02:52:05  19105  curl                  81 www.netflix.com
02:52:12  19111  curl                  17 www.netflix.com
02:52:19  19116  curl                   9 www.facebook.com
02:52:23  19118  curl                   3 www.facebook.com
```

Anwendungen - BCC



https://github.com/iovisor/bcc#tools 2019

Anwendungen

Beispiel Security eines WebServers

- eBPF ‘agent’ beobachtet eine Anwendung / Prozess
- Protokolliert was / mit wem dieser Kommuniziert
  - Viele clients am ’service port’
  - Datenbank
  - Schreibt logs, liest local fast nur aus ein paar Verzeichnissen
- Sobald sich das ‘gewohnte Schema’ ändert, wird Alarm ausgelöst
  - Server baut aktiv Verbindungen nach aussen auf oder schreibt lokal Files
- Läuft unabhängig vom Server und dessen Technik

Siehe Falco (https://falco.org/docs/) und andere

Fazit & Links

Im Vergleich zu vielen anderen Linux Kernel Features ist eBPF “neu”

Sowohl für ‘Admins’ als auch ‘Developer’ lohnt es sich die Features zu kennen.

Links:

“An eBPF overview” (technisch & detailliert): https://www.collabora.com/news-and-blog/blog/2019/04/05/an-ebpf-overview-part-1-introduction/

“eBPF Org”: (Community, Summit): https://ebpf.io

“eBPF Applications”: https://ebpf.io/applications/

“BCC”: https://github.com/iovisor/bcc

“bpftrace”: https://github.com/iovisor/bpftrace

Vielen Dank!

Gibt es noch Fragen ?