Understanding and using eBPF of the Linux kernel: Introduction to eBPF and related frameworks

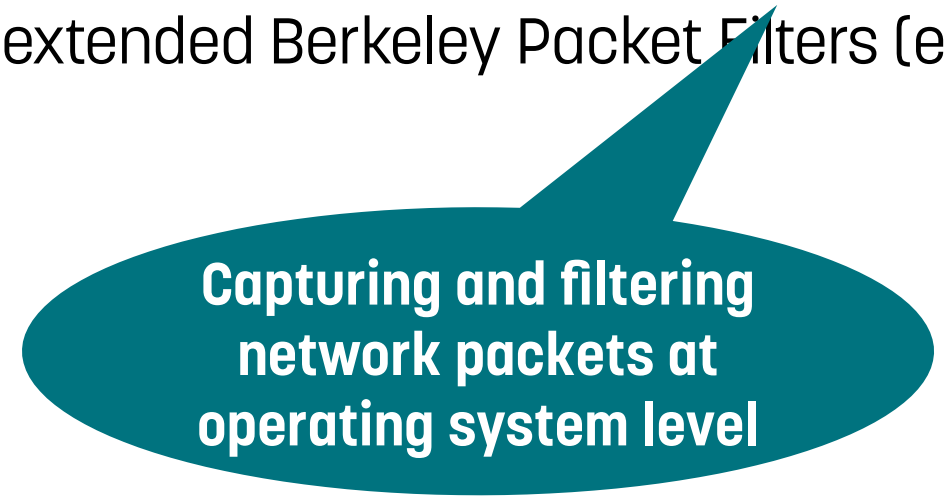
Open Source Automation Development Lab (OSADL) eG

What is eBPF?

- Evolution of the Berkeley Packet Filters
  - extended Berkeley Packet Filters (eBPF)

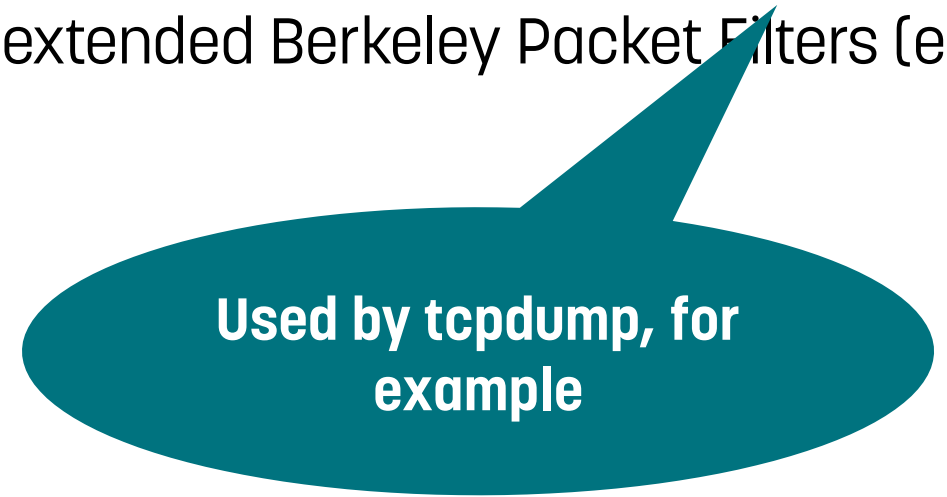
What is eBPF?

- Evolution of the Berkeley Packet Filters
  - extended Berkeley Packet Filters (eBPF)



What is eBPF?

- Evolution of the Berkeley Packet Filters
  - extended Berkeley Packet Filters (eBPF)



What is eBPF?

- Evolution of the Berkeley Packet Filters
  - extended Berkeley Packet Filters (eBPF)
- Technology for running sandboxed programs inside the Linux kernel
- Capabilities of the Linux kernel can be extended during runtime.

Use cases

Use cases

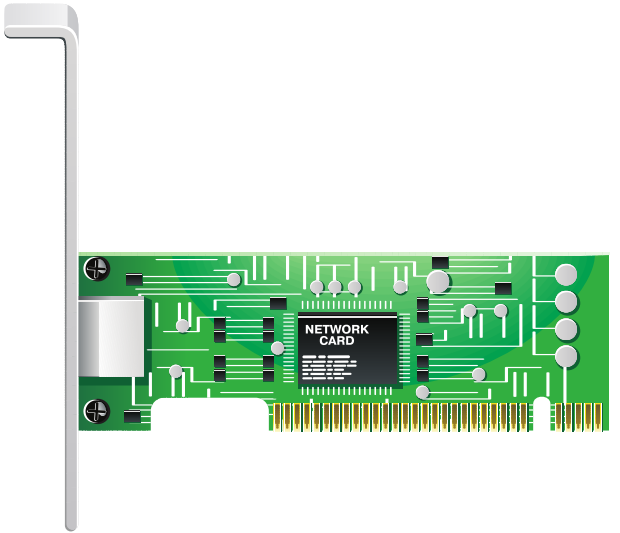
Monitoring

Security

Use cases

Monitoring

Security

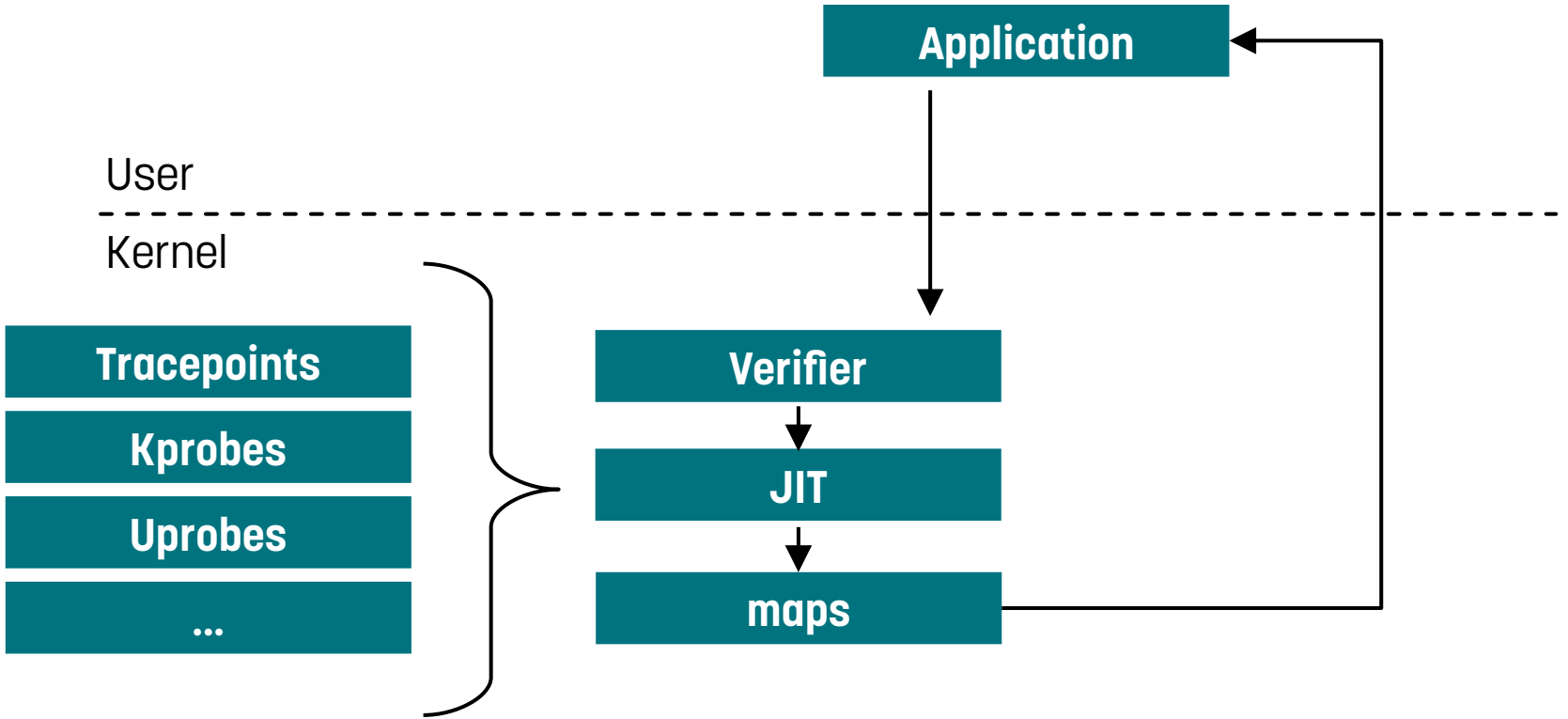


Networking

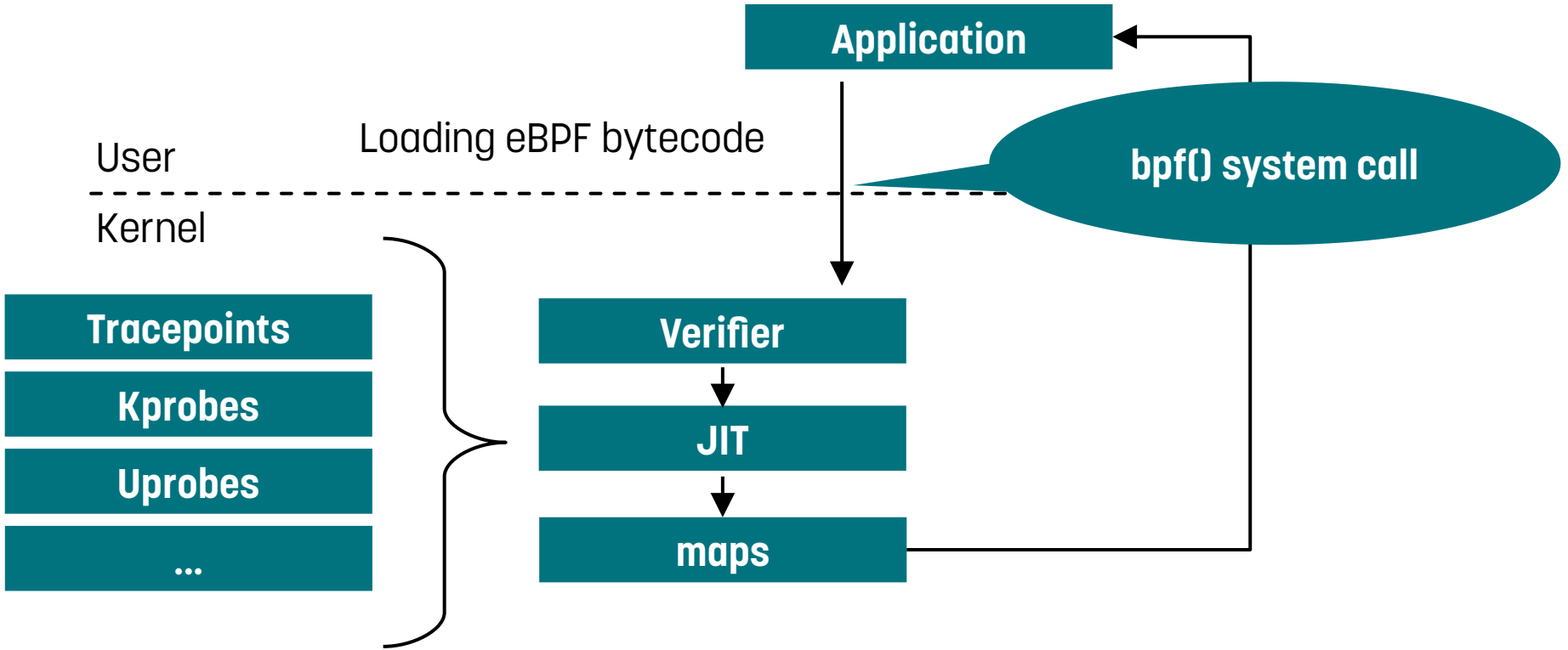
eBPF programs: Overview

- eBPF programs are event driven.
- They are attached to a specific hook:
  - Tracepoint
  - Kprobe
  - Uprobe
  - LSM hooks
  - ...

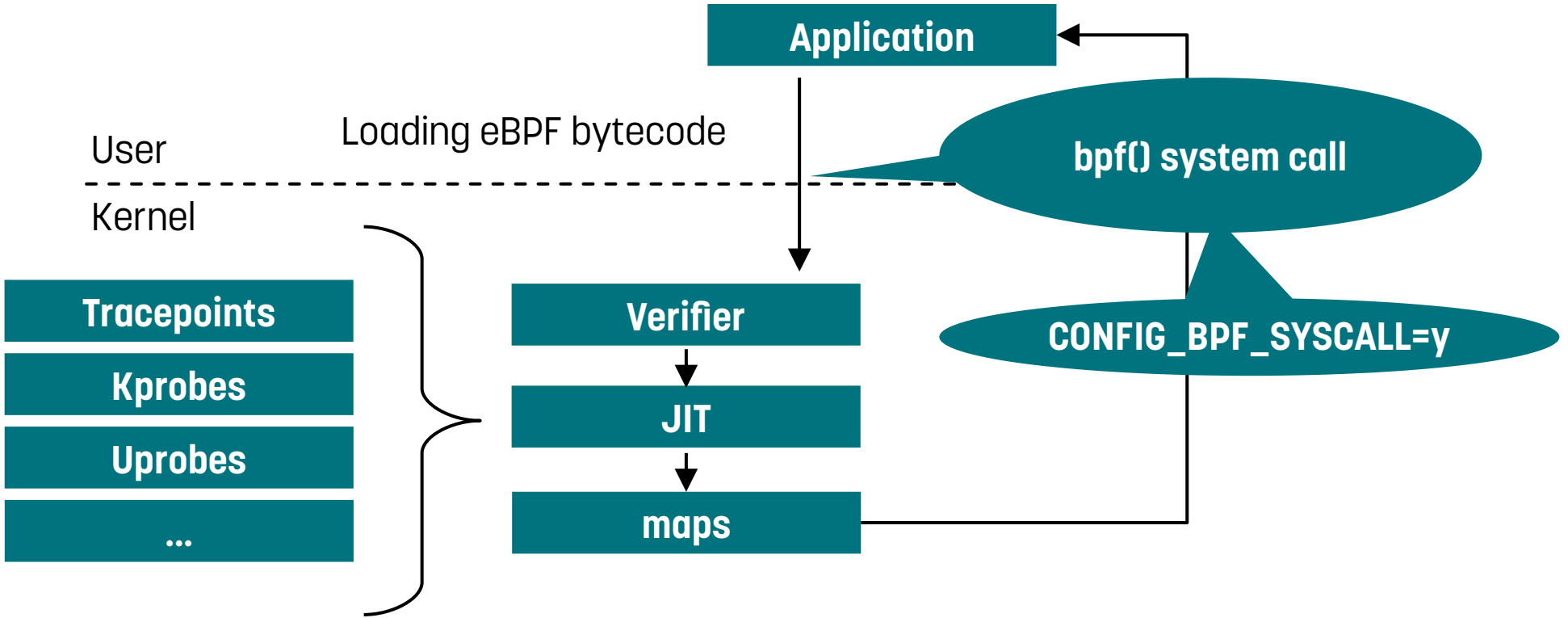
eBPF


Application
User
Kernel
Tracepoints
Kprobes
Uprobes
...
Verifier
JIT
maps

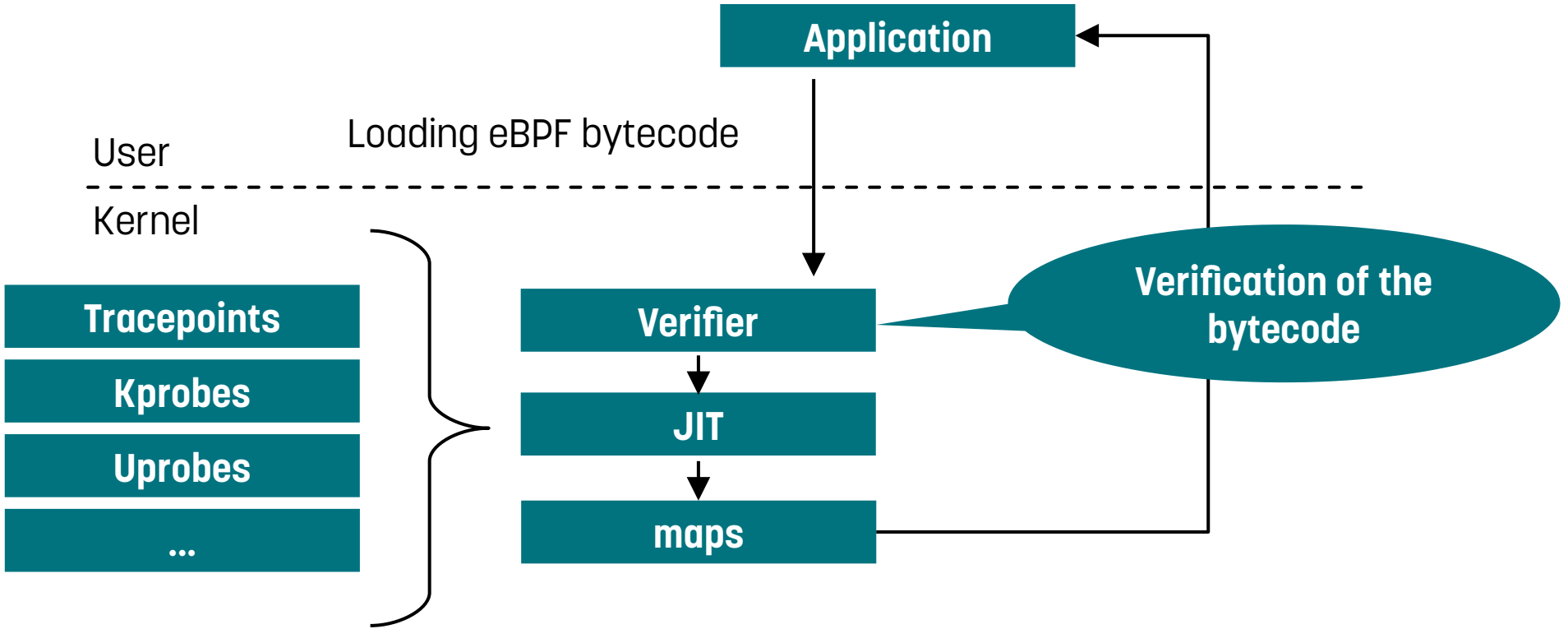
eBPF


Application
Loading eBPF bytecode
User
Kernel
bpf() system call
Tracepoints
Kprobes
Uprobes
...
Verifier
JIT
maps

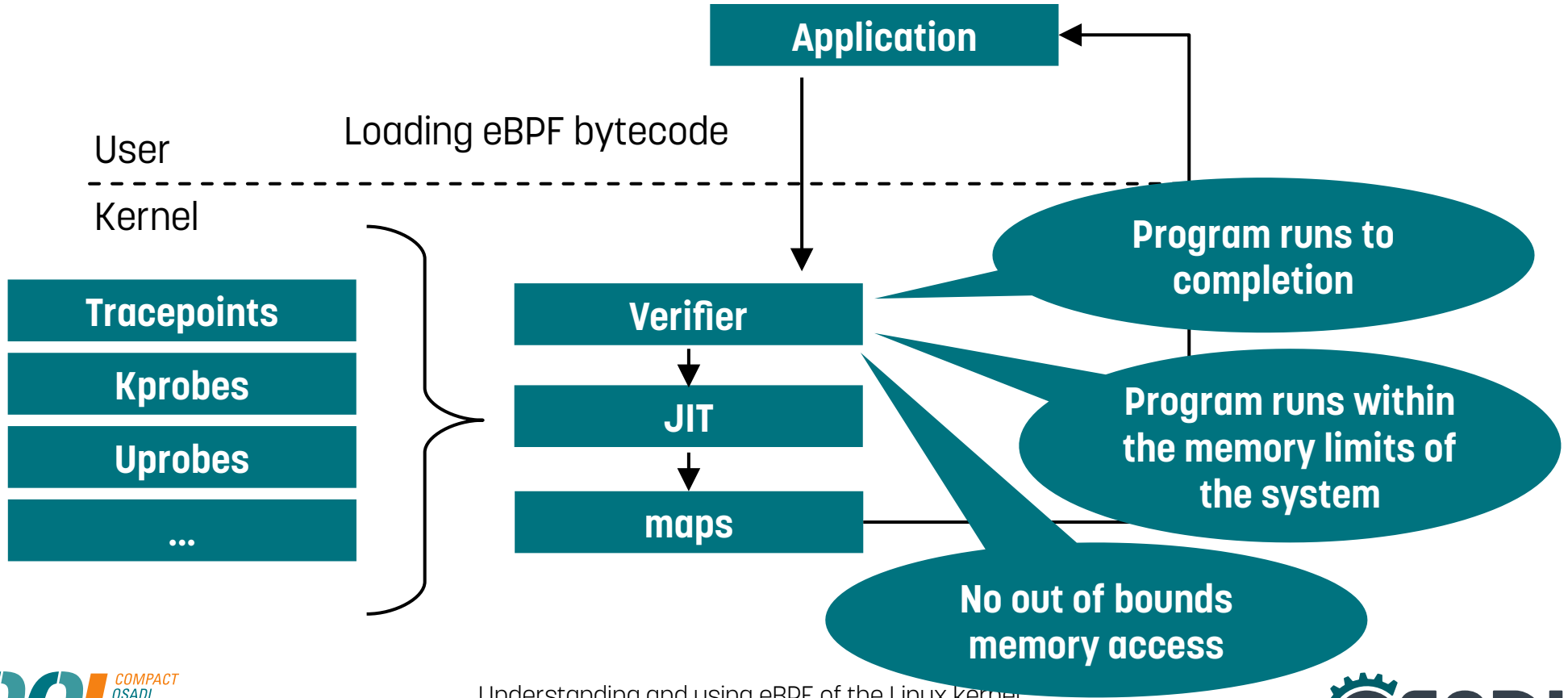
eBPF


Application
Loading eBPF bytecode
User
Kernel
bpf() system call
CONFIG_BPF_SYSCALL=y
Tracepoints
Kprobes
Uprobes
...
Verifier
JIT
maps

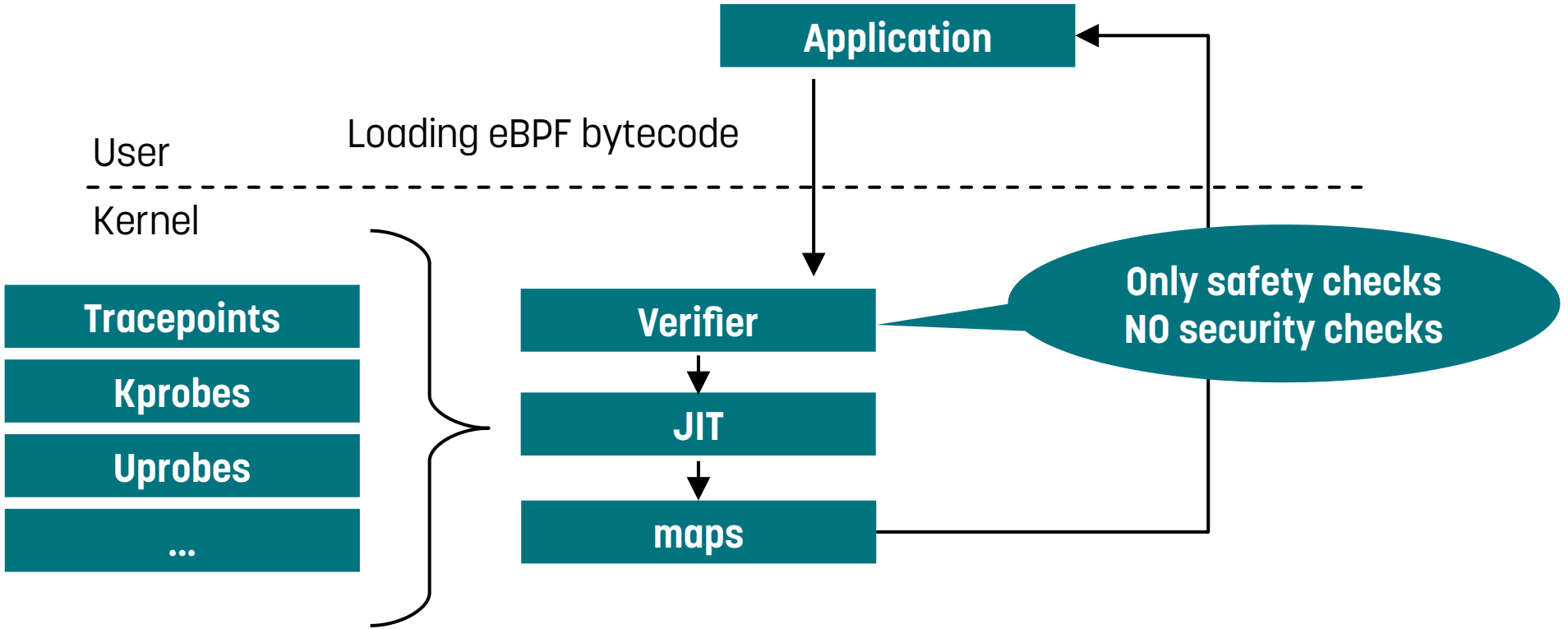
eBPF


Application
Loading eBPF bytecode
User
Kernel
Tracepoints
Kprobes
Uprobes
...
Verifier
JIT
maps
Verification of the bytecode

eBPF


Application
Loading eBPF bytecode
User
Kernel
Tracepoints
Kprobes
Uprobes
...
Verifier
JIT
maps
Program runs to completion
Program runs within the memory limits of the system
No out of bounds memory access

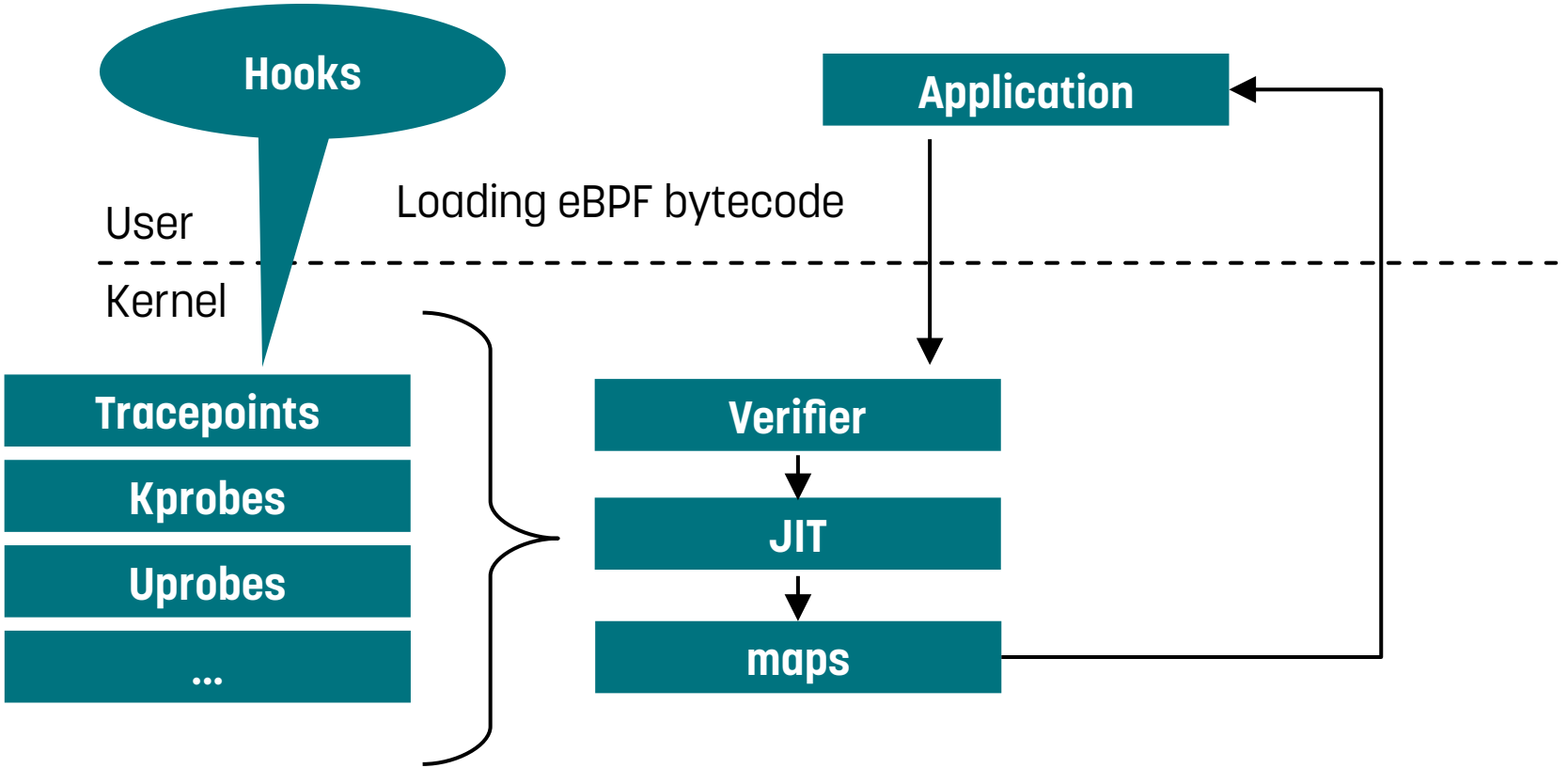
eBPF


Application
Loading eBPF bytecode
User
Kernel
Tracepoints
Kprobes
Uprobes
...
Verifier
JIT
maps
Only safety checks
NO security checks

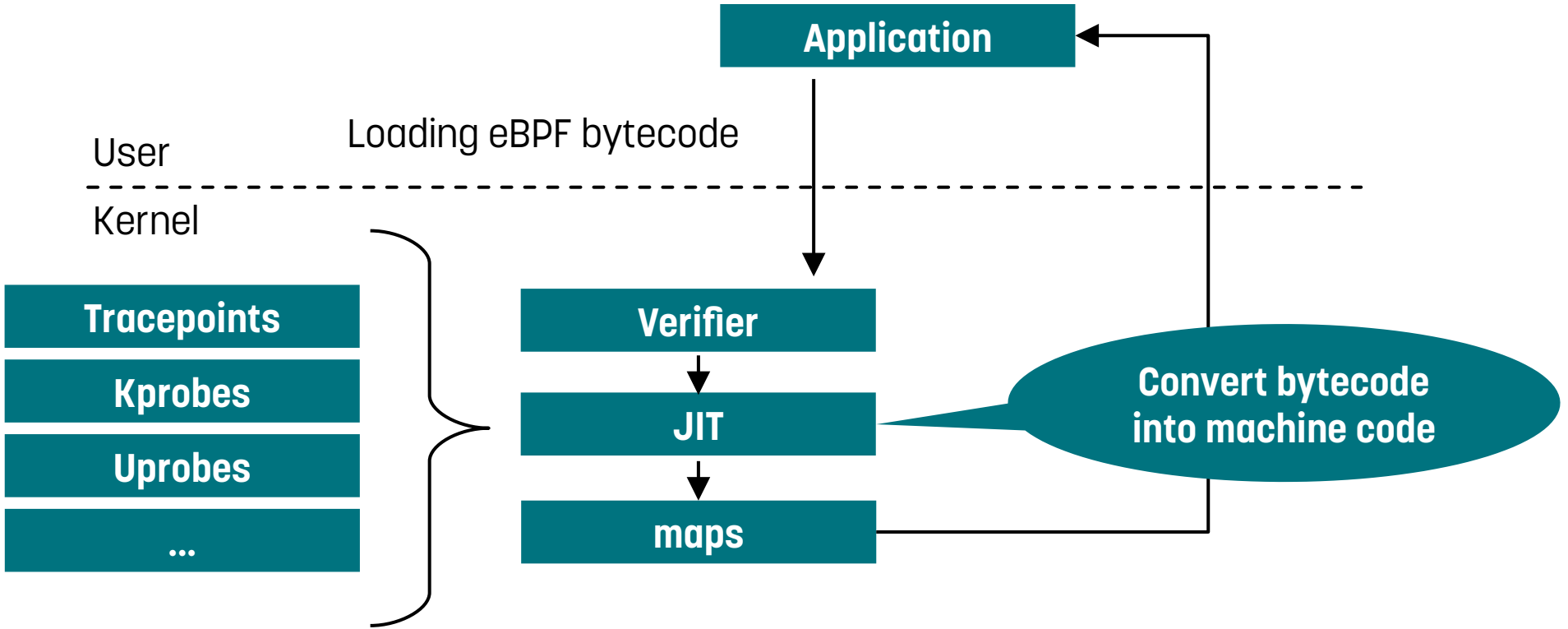
eBPF vs. kernel module

- Safe execution:
  - A **kernel module** can cause the entire system to crash
  - An **eBPF program** cannot cause the entire system to crash
- Flexibility:
  - **Kernel modules** have to be re-compiled for different kernel versions
  - **Kernel modules** usually need API changes for new kernel versions
  - An **eBPF program** can be used on different kernel versions

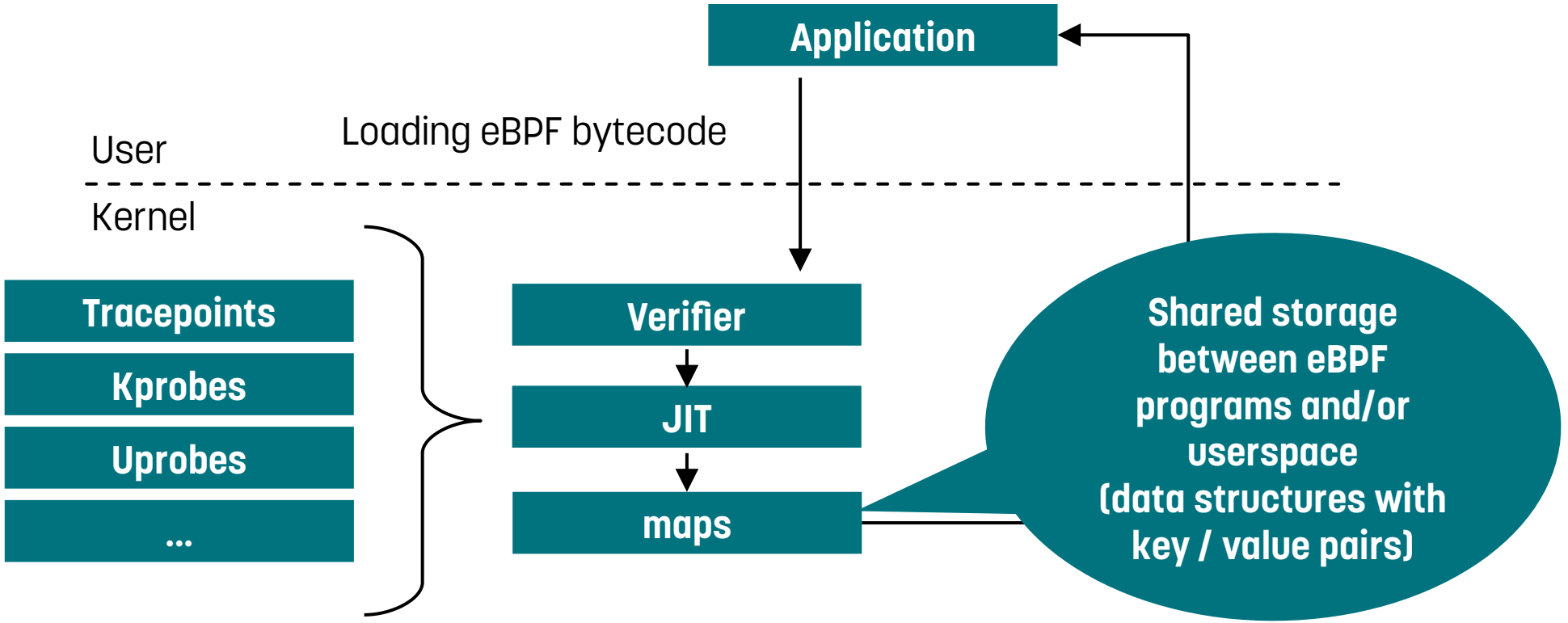
eBPF


Hooks
Application
Loading eBPF bytecode
User
Kernel
Tracepoints
Kprobes
Uprobes
...
Verifier
JIT
maps

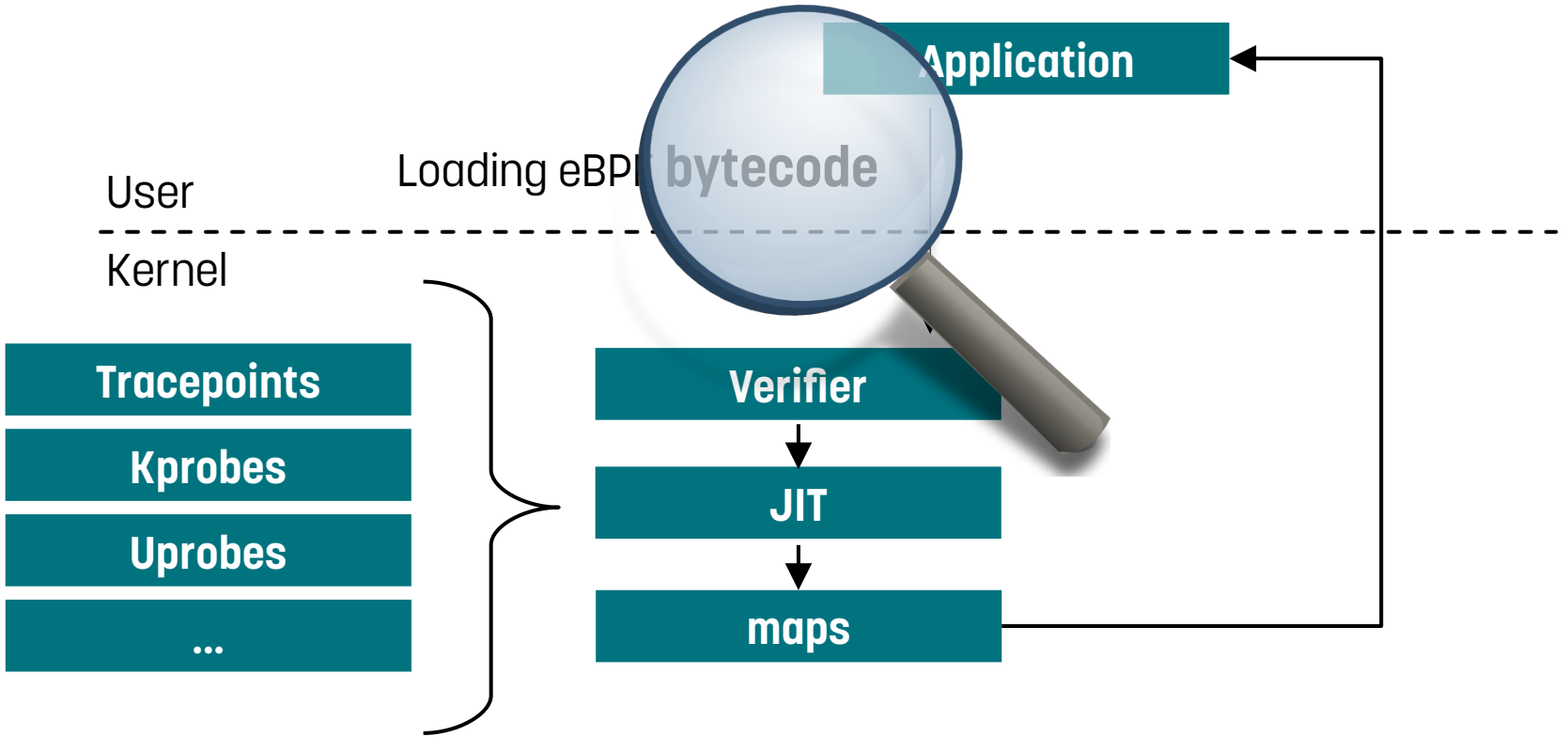
eBPF


Application
Loading eBPF bytecode
User
Kernel
Tracepoints
Kprobes
Uprobes
...
Verifier
JIT
maps
Convert bytecode into machine code

eBPF


Application
Loading eBPF bytecode
User
Kernel
Tracepoints
Kprobes
Uprobes
...
Verifier
JIT
maps
Shared storage between eBPF programs and/or userspace (data structures with key / value pairs)

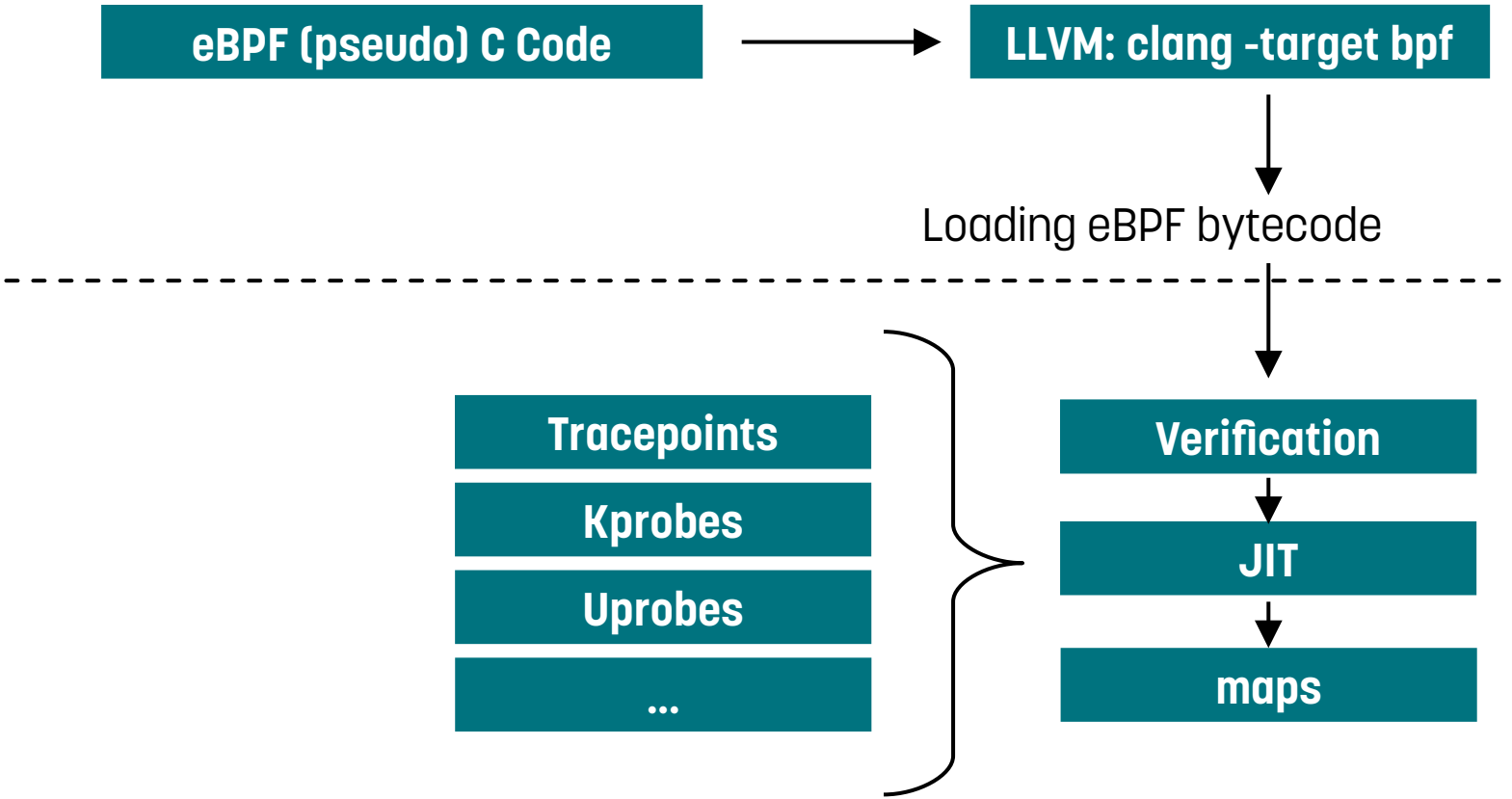
eBPF


Application
Loading eBPF bytecode
User
Kernel
Tracepoints
Kprobes
Uprobes
...
Verifier
JIT
maps

BPF bytecode

```
Disassembly of section tp/syscalls/sys_enter_read:

0000000000000000 <handle_tp>:
 0: b7 01 00 00 50 46 21 0a r1 = 169952848
 1: 63 1a f8 ff 00 00 00 00 *(u32 *)(r10 - 8) = r1
 2: 18 01 00 00 48 65 6c 6c 00 00 00 00 6f 20 65 42 r1 = 4784265842083521864 ll
 4: 7b 1a f0 ff 00 00 00 00 *(u64 *)(r10 - 16) = r1
 5: b7 01 00 00 00 00 00 00 r1 = 0
 6: 73 1a fc ff 00 00 00 00 *(u8 *)(r10 - 4) = r1
 7: bf a1 00 00 00 00 00 00 r1 = r10
 8: 07 01 00 00 f0 ff ff ff r1 += -16
 9: b7 02 00 00 0d 00 00 00 r2 = 13
10: 85 00 00 00 06 00 00 00 call 6
11: b7 00 00 00 00 00 00 00 r0 = 0
12: 95 00 00 00 00 00 00 00 exit
```

eBPF and LLVM



Prepare the kernel

```
CONFIG_HAVE_EBPF_JIT=y
CONFIG_BPF=y
CONFIG_BPF_SYSCALL=y
CONFIG_BPF_JIT=y
CONFIG_BPF_EVENTS=y
```

Install the tools (on Debian Bookworm)

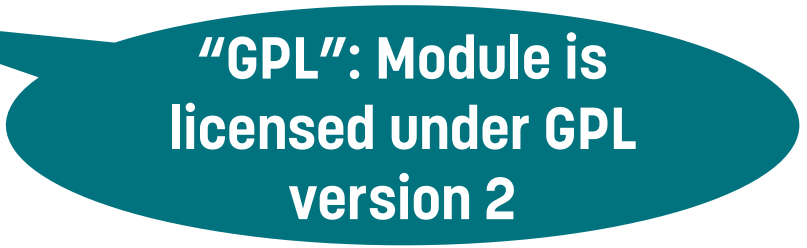
```
apt-get install llvm \
                gcc-multilib \
                libbpf-dev \
                linux-headers-$(uname -r)

apt install -t bookworm-backports bpftool
```

eBPF and LLVM

```
/* hello_epf.c */

#define BPF_NO_GLOBAL_DATA
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>

char LICENSE[] SEC("license") = "GPL";

SEC("tp/syscalls/sys_enter_read")
int handle_tp(void *ctx)
{
  bpf_printk("Hello eBPF!\n");
  return 0;
}
```

eBPF and LLVM

```
/* hello_epf.c */

#define BPF_NO_GLOBAL_DATA
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>

char LICENSE[] SEC("license") = "GPL";

SEC("tp/syscalls/sys_enter_read")
int handle_tp(void *ctx)
{
  bpf_printk("Hello eBPF!\n");
  return 0;
}
```

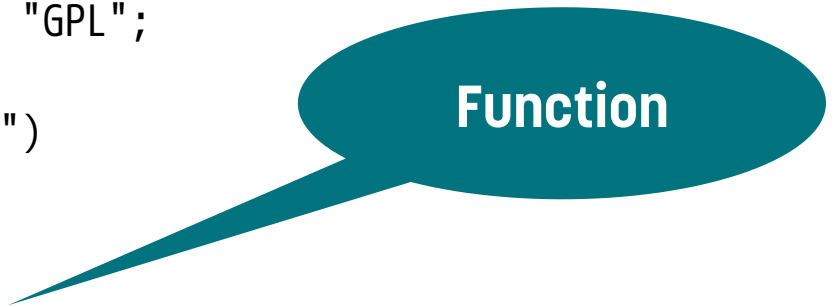


eBPF and LLVM

```
/* hello_epf.c */

#define BPF_NO_GLOBAL_DATA
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>

char LICENSE[] SEC("license") = "GPL";

SEC("tp/syscalls/sys_enter_read")
int handle_tp(void *ctx)
{
  bpf_printk("Hello eBPF!\n");
  return 0;
}
```

eBPF and LLVM

```
/* hello_epf.c */

#define BPF_NO_GLOBAL_DATA
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>

char LICENSE[] SEC("license") = "GPL";

SEC("tp/syscalls/sys_enter_read")
int handle_tp(void *ctx)
{
  bpf_printk("Hello eBPF!\n");
  return 0;
}
```

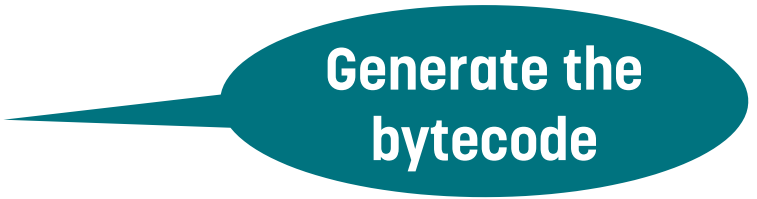


eBPF and LLVM

```
# clang -O2 \
        -target bpf \
        -c hello_ebpf.c \
        -o hello_ebpf.o

# echo 1 > /sys/kernel/debug/tracing/events/syscall/sys_enter_read/enable
# bpftool prog load \
     hello_ebpf.o \
     /sys/fs/bpf/hello_ebpf \
     autoattach

# cat /sys/kernel/debug/tracing/trace_pipe
```

eBPF and LLVM

```
# clang -O2 \
        -target bpf \
        -c hello_ebpf.c \
        -o hello_ebpf.o
```




```
# echo 1 > /sys/kernel/debug/tracing/events/syscall/sys_enter_read/enable
# bpftool prog load \
     hello_ebpf.o \
     /sys/fs/bpf/hello_ebpf \
     autoattach

# cat /sys/kernel/debug/tracing/trace_pipe
```

eBPF and LLVM

```
$ objdump -x hello_ebpf.o

[...]
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
[...]
  1 tp/syscalls/sys_enter_read 00000060  0000000000000000  0000000000000000  00000040  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 license       00000004  0000000000000000  0000000000000000  000000a0  2**0
                  CONTENTS, ALLOC, LOAD, DATA
[...]
SYMBOL TABLE:
[...]
0000000000000000 g     F tp/syscalls/sys_enter_read 0000000000000060 handle_tp
0000000000000000 g     O license  0000000000000004 LICENSE
```

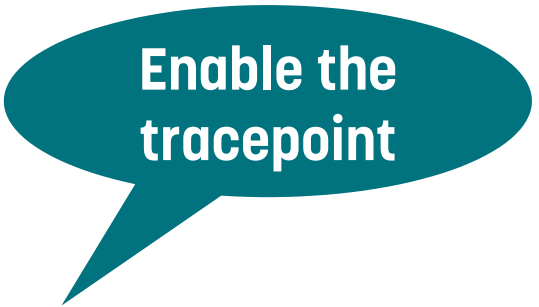
eBPF and LLVM

```
# clang -O2 \
        -target bpf \
        -c hello_ebpf.c \
        -o hello_ebpf.o

# echo 1 > /sys/kernel/debug/tracing/ev
# bpftool prog load \
     hello_ebpf.o \
     /sys/fs/bpf/hello_ebpf \
     autoattach

# cat /sys/kernel/debug/tracing/trace_pipe
```

If compilation fails with asm/types.h missing on a Debian system, this can be fixed by installing the gcc-multilib package

eBPF and LLVM

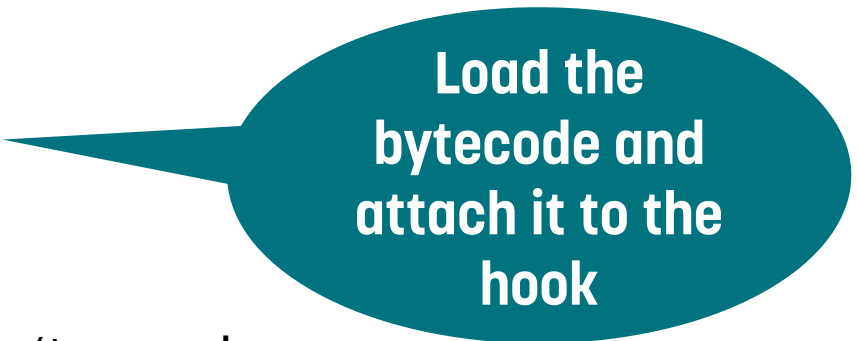
```
# clang -O2 \
        -target bpf \
        -c hello_ebpf.c \
        -o hello_ebpf.o
```




```
# echo 1 > /sys/kernel/debug/tracing/events/syscall/sys_enter_read/enable
# bpftool prog load \
     hello_ebpf.o \
     /sys/fs/bpf/hello_ebpf \
     autoattach

# cat /sys/kernel/debug/tracing/trace_pipe
```

eBPF and LLVM

```
# clang -O2 \
        -target bpf \
        -c hello_ebpf.c \
        -o hello_ebpf.o

# echo 1 > /sys/kernel/debug/tracing/events/syscall/sys_enter_read/enable
# bpftool prog load \
     hello_ebpf.o \
     /sys/fs/bpf/hello_ebpf \
     autoattach

# cat /sys/kernel/debug/tracing/trace_pipe
```



eBPF and LLVM

```
# clang -O2 \
        -target bpf \
        -c hello_ebpf.c \
        -o hello_ebpf.o

# echo 1 > /sys/kernel/debug/tracing/events/syscall/sys_enter_read/enable
# bpftool prog load \
     hello_ebpf.o \
     /sys/fs/bpf/hello_ebpf \
     autoattach

# cat /sys/kernel/debug/tracing/trace_pipe
```

eBPF and LLVM

```
# cat /sys/kernel/debug/tracing/trace_pipe

lxterminal-3451    [003] d..31  9960.396951: bpf_trace_printk: Hello eBPF!

cat-156618  [001] d..31  9960.396954: bpf_trace_printk: Hello eBPF!

cat-156618  [001] d..31  9960.396961: bpf_trace_printk: Hello eBPF!

cat-156618  [001] d..31  9960.396965: bpf_trace_printk: Hello eBPF!
```

eBPF features

- The eBPF feature set is continuously growing.
- An overview of new features available in a specific kernel version can be found here:

https://docs.ebpf.io/linux/timeline/

Get a list of loaded eBPF programs

```
# bpftool prog list
[...]
112: cgroup_device  tag 0b1566e4b83190c5  gpl
loaded_at 2025-02-06T14:48:49+0100  uid 0
xlated 560B  jited 352B  memlock 4096B
pids systemd(1)
118: tracepoint  name handle_tp  tag eb64ccfb0c7075ea  gpl
loaded_at 2025-02-06T14:48:53+0100  uid 0
xlated 104B  jited 68B  memlock 4096B  map_ids 23
```

Get a list of loaded eBPF programs

```
# bpftool prog list
[...]
112: cgroup_device  tag 0b1566e4b83190c5  gpl
loaded_at 2025-02-06T14:48:49+0100  uid 0
xlated 560B  jited 352B  memlock 4096B
pids systemd(1)
118: tracepoint  name handle_tp  tag eb64ccfb0c7075ea  gpl
loaded_at 2025-02-06T14:48:53+0100  uid 0
xlated 104B  jited 68B  memlock 4096B  map_ids 23
```

Get a list of loaded eBPF programs

```
# bpftool prog list
[...]
112: cgroup_device  tag 0b1566e4b83190c5  gpl
loaded_at 2025-02-06T14:..                  uid 0
xlated 560B  jited 352B
pids systemd(1)
118: tracepoint  name handle_tp  tag eb64ccfb0c7075ea  gpl
loaded_at 2025-02-06T14:48:53+0100  uid 0
xlated 104B  jited 68B  memlock 4096B  map_ids 23
```

program type

Program types

```
/* linux/include/uapi/linux/bpf.h */
enum bpf_prog_type {
        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,
[...]
```

Let's try something bad

```
/* hello_epf.c */
[...]

SEC("tp/syscalls/sys_enter_read")
int handle_tp(void *ctx)
{
  bpf_printk("Hello eBPF!\n");
  return 0;
}
```

Let's try something bad

```
/* hello_epf.c */
[...]

SEC("tp/syscalls/sys_enter_read")
int handle_tp(void *ctx)
{
  while (1)
     bpf_printk("Hello eBPF!\n");
  return 0;
}
```

Let's try something bad

```
# clang -O2 \
        -target bpf \
        -c hello_ebpf.c \
        -o hello_ebpf.o

# echo 1 > /sys/kernel/debug/tracing/events/syscall/sys_enter_read/enable
# bpftool prog load \
     hello_ebpf.o \
     /sys/fs/bpf/hello_ebpf \
     autoattach
```

Let's try something bad

```
[...]
10: (85) call bpf_trace_printk#6      ; R0_w=scalar()
11: (05) goto pc-8
infinite loop detected at insn 4
processed 20 insns (limit 1000000) max_states_per_insn 0
total_states 1 peak_states 1 mark_read 1
-- END PROG LOAD LOG --
libbpf: prog 'handle_tp': failed to load: -22
libbpf: failed to load object 'hello_ebpf.o'
Error: failed to load object file
```

eBPF and functions

```
[...]
SEC("tp/syscalls/sys_enter_read")
int handle_tp(void *ctx)
{
  bpf_printk("Hello eBPF!\n");
  return 0;
}
```

eBPF and functions

```
[...]
void my_helper_function(void)
{
  bpf_printk("Hello eBPF!\n");
}

SEC("tp/syscalls/sys_enter_read")
int handle_tp(void *ctx)
{
  my_helper_function();
  return 0;
}
```

eBPF helper functions

- eBPF programs cannot call arbitrary kernel functions.
- eBPF can use various helper functions (man bpf-helpers):
  - Generate random numbers: bpf_get_prandom_u32()
  - Get time since system was booted: bpf_ktime_get_ns()
  - Get process context: bpf_get_current_task ()
  - Get the name of the current task: bpf_get_current_comm(void *buf, u32 size_of_buf)
  - ...

eBPF helper functions

- eBPF programs cannot call arbitrary kernel functions.
- eBPF can use various helper functions (aka bpf-helpers):
  - Generate random numbers: bpf_get_prandom_u32()
  - Get time since system boot: bpf_ktime_get_ns()
  - Get process ...
  - Get the name of the current task: bpf_get_current_comm(void *buf, u32 size_of_buf)
  - ...

Linux 5.13 gained support for so called “kfuncs”

kfuncs

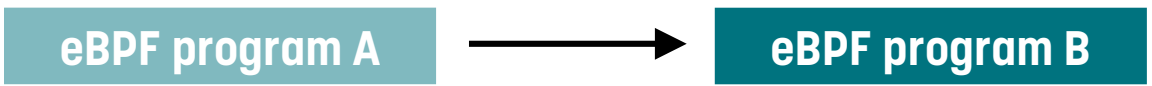
- Kernel functions can be annotated to be called by an eBPF program.
- Kfuncs are usually limited to specific program types.
- Be careful: The API of kernel functions can change!

Tail calls

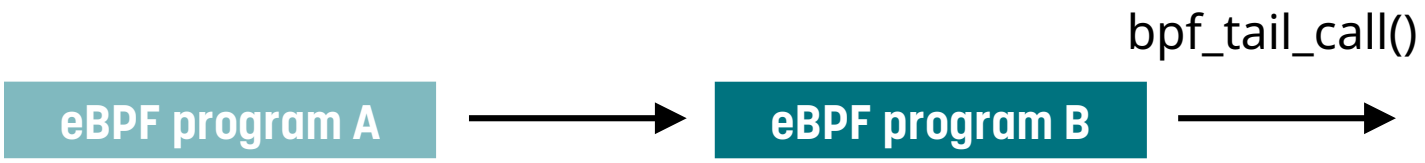
eBPF program A

Tail calls

Tail calls



Tail calls

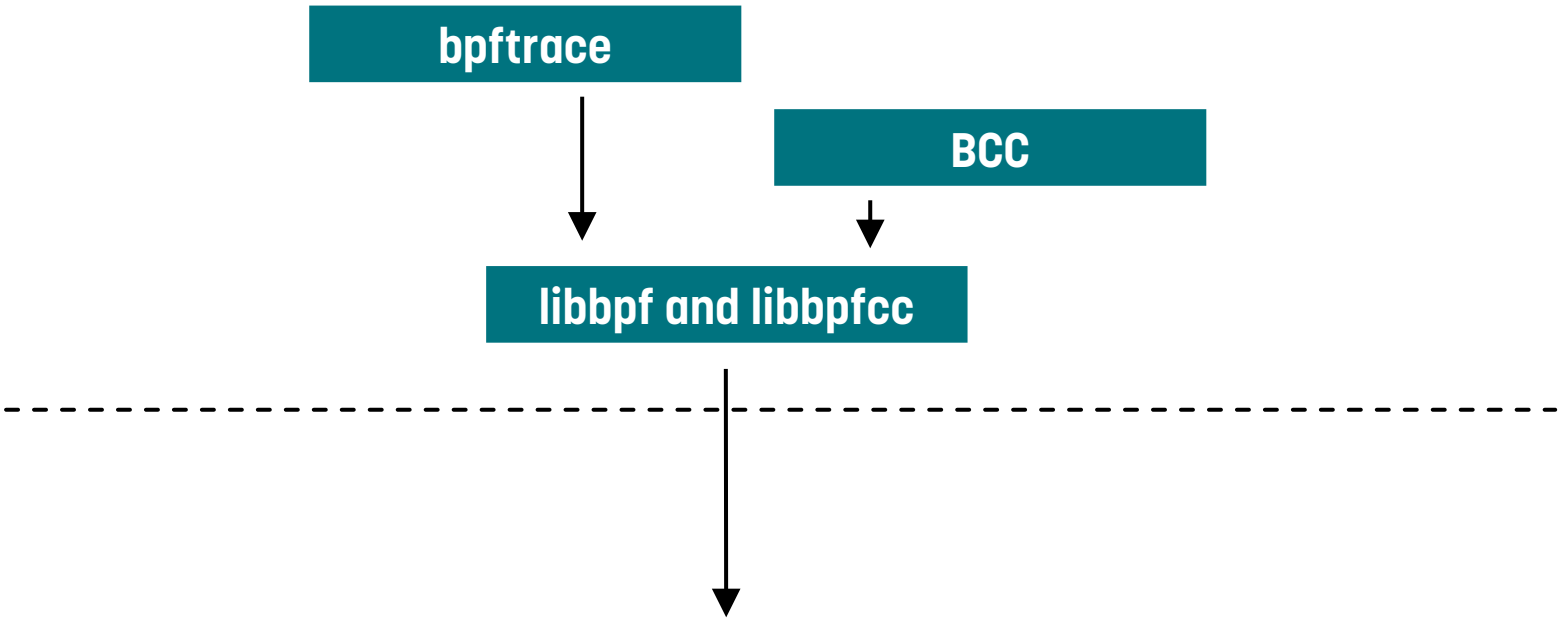


Tail calls

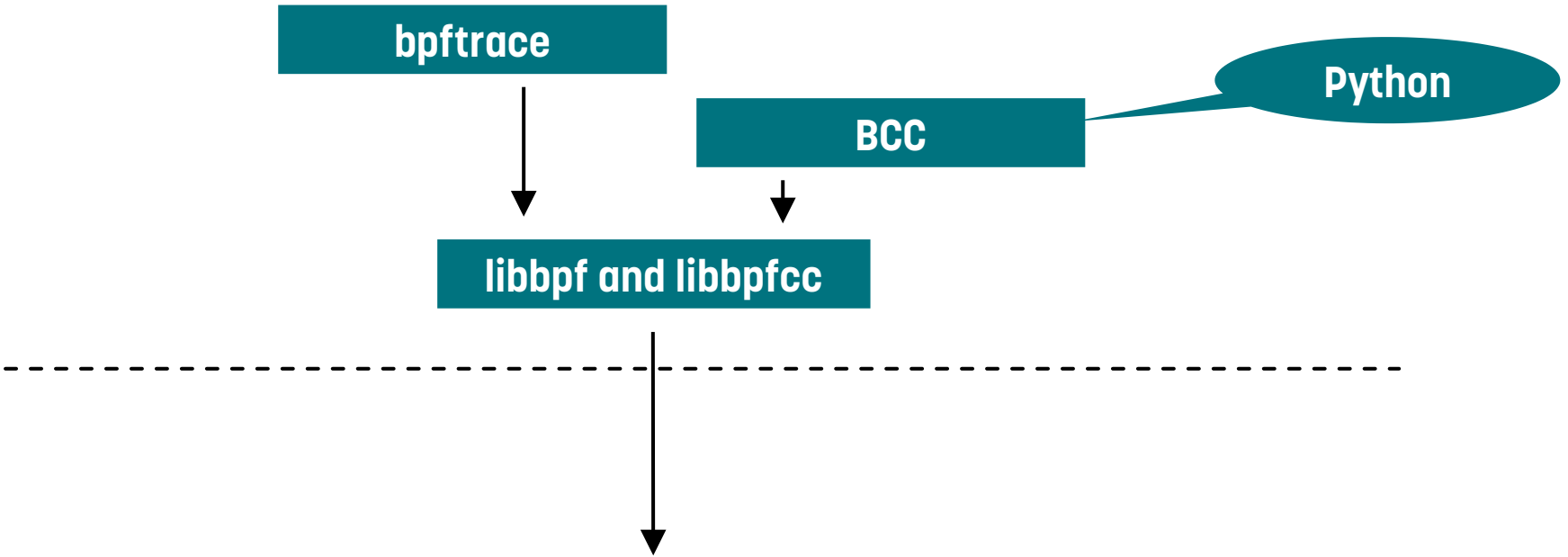
Tail calls

tail calls are comparable to execve() in userspace

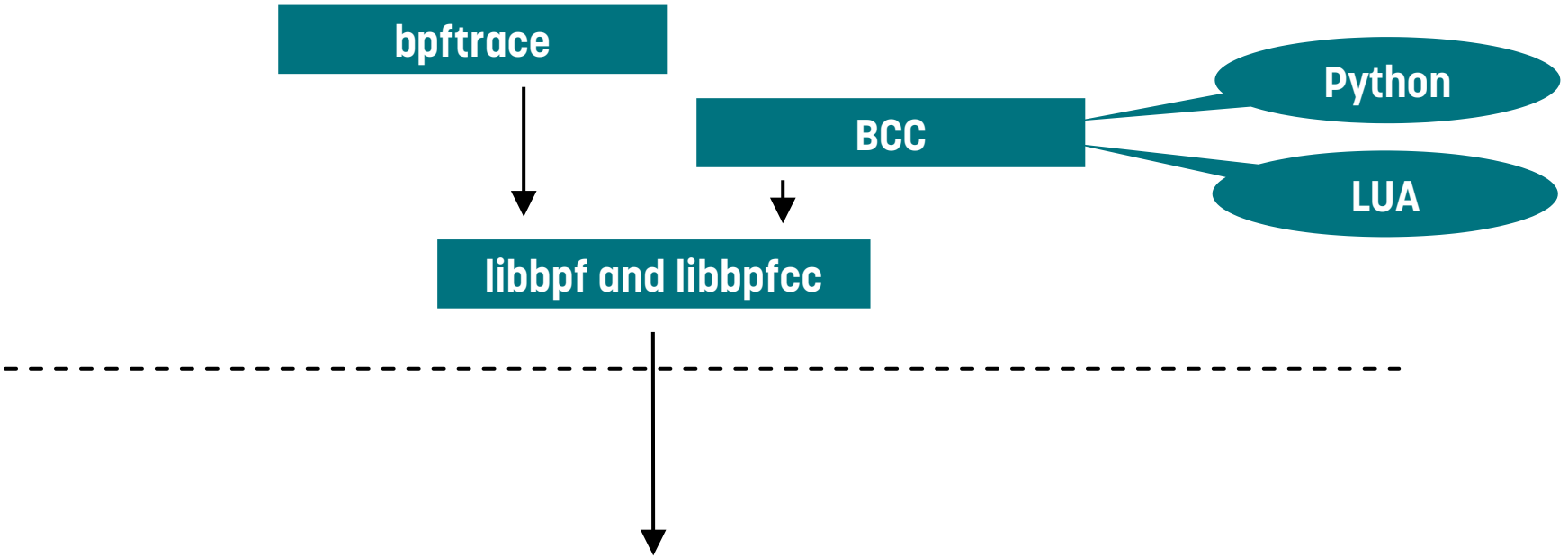
Tooling landscape



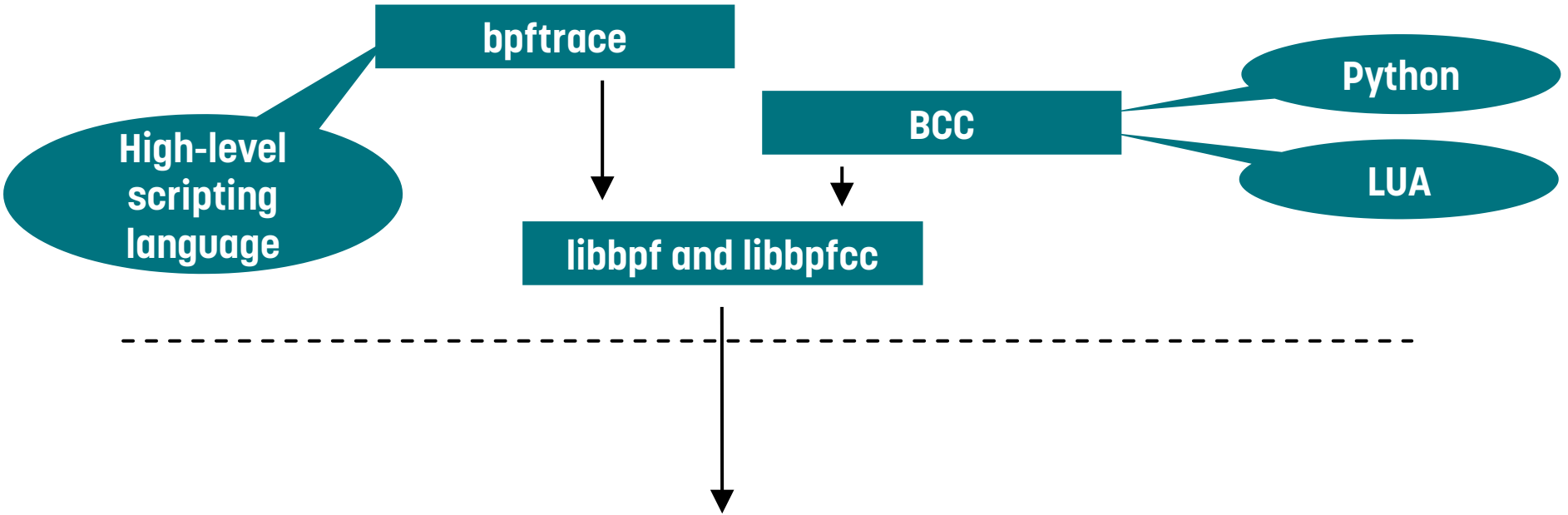
Tooling landscape



Tooling landscape



Tooling landscape



BPF Compiler Collection (BCC)

- Tool collection and library for creating eBPF programs
- Contains a C wrapper around LLVM
- Offers frontends for Python and LUA
- Includes various examples

BPF Compiler Collection (BCC)

- Tool collection and library for creating eBPF programs
- Contains a C wrapper around LLVM
- Offers frontends for Python and LUA
- Includes various examples

apt-get install bpfcc-tools

BCC

```
/* handle_tp.c */

#include <uapi/linux/ptrace.h>

int handle_tp(void *ctx)
{
    bpf_trace_printk("Hello eBPF!\n");
    return 0;
}
```

BCC

```
#!/usr/bin/python
# load_ebpf.py
from bcc import BPF

mybpf = BPF(src_file="handle_tp.c")
mybpf.attach_tracepoint(tp="syscalls:sys_enter_read",
fn_name="handle_tp")

mybpf.trace_print()
```

BCC

```
# ./load_epf.py


b'      lxterminal-3051    [002] d..31 28602.801927:
bpf_trace_printk: Hello eBPF!'
b''
b'      lxterminal-3051    [002] d..31 28602.801929:
bpf_trace_printk: Hello eBPF!'
b''
b'      lxterminal-3051    [002] d..31 28602.801933:
bpf_trace_printk: Hello eBPF!'
```

BCC (python)

```
#!/usr/bin/python

from bcc import BPF

my_program = """
int hello_ebpf(void *context) {
    bpf_trace_printk("eBPF is COOL :)");
    return 0;
}
"""

mybpf = BPF(text=my_program)
system_call = mybpf.get_syscall_fnname("read")
mybpf.attach_kprobe(event=system_call, fn_name="hello_epf")

mybpf.trace_print()
```

BCC (python)

```
#!/usr/bin/python

from bcc import BPF

my_program = """
int hello_ebpf(void *context) {
    bpf_trace_printk("eBPF is COOL :)");
    return 0;
}
"""

mybpf = BPF(text=my_program)
system_call = mybpf.get_syscall_fnname("read")
mybpf.attach_kprobe(event=system_call, fn_name="hello_epf")

mybpf.trace_print()
```

BCC (python)

```
#!/usr/bin/python

from bcc import BPF

my_program = """
int hello_ebpf(void *context) {
    bpf_trace_printk("eBPF is COOL :)");
    return 0;
}
"""

mybpf = BPF(text=my_program)
system_call = mybpf.get_syscall_fnname("read")
mybpf.attach_kprobe(event=system_call, fn_name="hello_epf")

mybpf.trace_print()
```

BCC (python)

```
#!/usr/bin/python

from bcc import BPF

my_program = """
int hello_ebpf(void *context) {
    bpf_trace_printk("eBPF is COOL :)");
    return 0;
}
"""

mybpf = BPF(text=my_progra
system_call = mybpf.get_s
mybpf.attach_kprobe(event

mybpf.trace_print()
```

Read trace_pipe and print the output

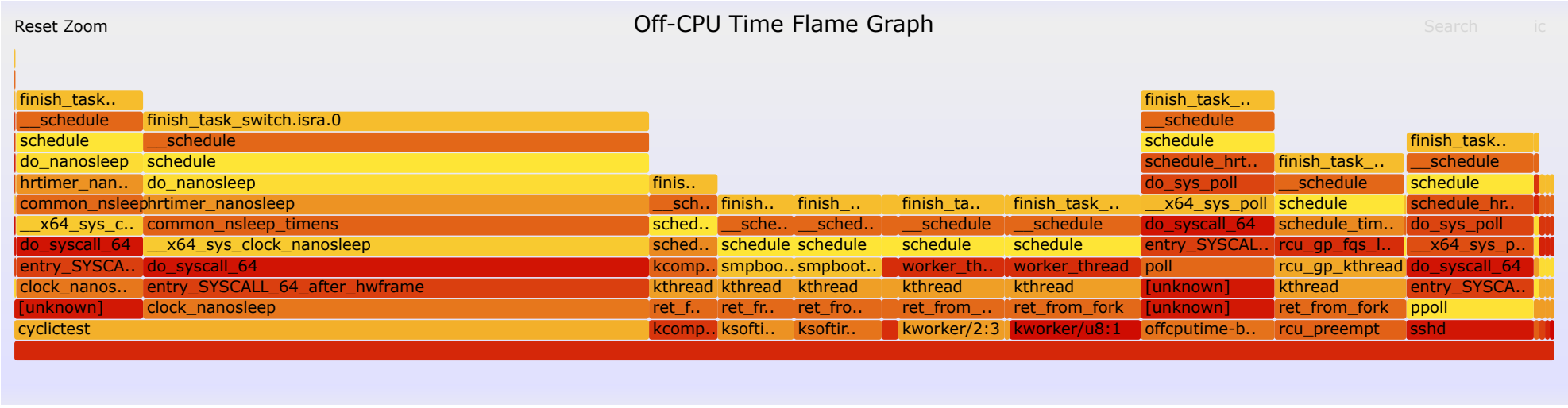
bpfcc-tools

```
# dpkg -L bpfcc-tools
/usr/sbin/argdist-bpfcc
/usr/sbin/bashreadline-bpfcc
/usr/sbin/bindsnoop-bpfcc
[...]
/usr/sbin/offcputime-bpfcc
```

bpfcc-tools: offcputime

```
# git clone \
  https://github.com/brendangregg/FlameGraph.git

# offcputime-bpfcc  -f 5 | \
  ./FlameGraph/flamegraph.pl --bgcolors=blue \
  --title="Off-CPU Time Flame Graph" > out.svg
```

bpfcc-tools: offcputime



bpftrace

- High-level tracing language for eBPF
- Inspired by AWK and C
- Built on top of LLVM and BCC
- Provides an easy way for writing eBPF programs that interact with the tracing infrastructure
- Comes with various examples

bpftrace

- High-level tracing language for eBPF
- Inspired by AWK and C
- Built on top of LLVM and C
- Provides an easy way of writing eBPF programs that interact with the tracing infrastructure
- **apt-get install bpftrace**

bpftrace

```
#!/usr/bin/bpftrace

tracepoint:syscalls:sys_enter_read
{
    printf("Hello eBPF\n");
}
```

bpftrace

```
#!/usr/bin/bpftrace

tracepoint:syscalls:sys_enter_read
{
    printf("Hello eBPF\n");
}
```

bpftrace

```
#!/usr/bin/bpftrace

tracepoint:syscalls:sys_enter_read
{
    printf("Hello eBPF\n");
}
```

bpftrace

```
#!/usr/bin/bpftrace

tracepoint:syscalls:sys_enter_read
{
    printf("Hello eBPF\n");
}
```

bpftrace examples

```
# ls -l /usr/sbin/*.bt
/usr/sbin/bashreadline.bt
[...]
/usr/sbin/biosnoop.bt
/usr/sbin/cpuwalk.bt
[...]
/usr/sbin/vfscount.bt
/usr/sbin/vfsstat.bt
/usr/sbin/writeback.bt
/usr/sbin/xfsdist.bt
```

bpftrace: bashreadline.bt

```
# bashreadline.bt

Attaching 2 probes...
Tracing bash commands... Hit Ctrl-C to end.
TIME      PID    COMMAND
11:58:06  2448   ls
11:58:10  2448   ps
11:58:19  2448   cat /etc/fstab
11:58:28  2448   dmesg
```

bpftrace: cpuwalk.bt

```
# cpuwalk.bt
Attaching 2 probes...
Sampling CPU at 99hz... Hit Ctrl-C to end.
^C

@cpu:
[0, 1)               1735 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[1, 2)               1662 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   |
[2, 3)                  1 |                                                    |
[3, 4)                  3 |                                                    |
```

bpftrace: runqlen.bt

```
# runqlen.bt
Attaching 2 probes...
Sampling run queue length at 99 Hertz... Hit Ctrl-C to end.
^C

@runqlen:
[0, 1)              1309 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[1, 2)               140 |@@@@@                                               |

[...]

[96, 97)               0 |                                                    |
[97, 98)               0 |                                                    |
[98, 99)               1 |                                                    |
[99, 100)              0 |                                                    |
[100, ...)            84 |@@@                                                 |
```