



Embedded Linux
Conference

ply: lightweight eBPF tracing

Frank Vasquez / Lunar Energy

#lfelc @st8l3ss

About me

- Over a decade doing embedded Linux
- Networked audio, sonar and LoRa
- MELP3 co-author
- Home electrification
- Aspiring SRE



Agenda

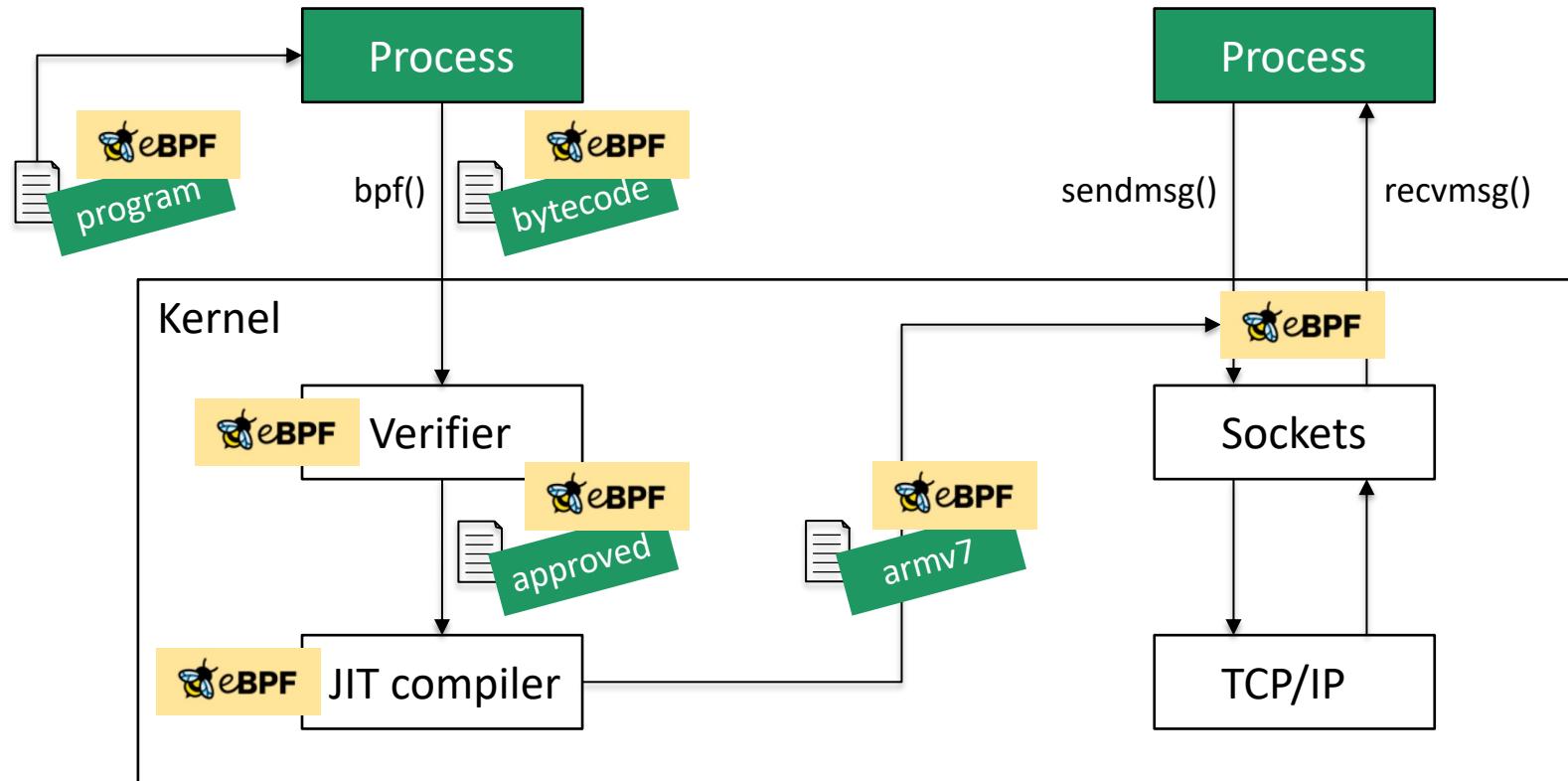
- What is eBPF?
- Why ply and not bpftrace?
- What ply can and can't do
- Enabling eBPF in the kernel
- Adding ply to a Buildroot image
- Example ply scripts



What is eBPF?

- Kernel feature (introduced in Linux 3.18)
- Sandbox environment
- Verification and JIT compilation
- Event-driven runtime
- Low overhead
- Observability, networking and security

How does eBPF work?



Who uses eBPF?

- Netflix for observability
- Facebook for load balancing
- Google for GKE Dataplane V2
- AWS for Bottlerocket
- Microsoft for Windows
- Pixie Labs acquired by New Relic



Leading eBPF projects

- BCC – kernel tracing toolkit and library
- bpftrace – high-level tracing language
- Katran – high-performance load balancer
- Cilium – operating & securing K8s clusters
- Falco – security monitoring tool



Why not bpftrace?

- bpftrace depends on LLVM at runtime
- bpftrace depends on BCC at runtime
- BCC depends on LLVM for compilation
- BCC only targets select 64-bit architectures
- BCC requires kernel sources at runtime



Why ply?

- ply only depends on libc at runtime
- ply targets 32-bit arm and powerpc
- ply is easy to port to more architectures
- ply is included in Buildroot 2021.02 LTS
- ply syntax is very similar to bpftrace



Dynamic vs static instrumentation

- Dynamic
 - Insert breakpoints at instruction addresses
 - Record information then continue execution
 - Susceptible to interface instability and inlining
- Static
 - Stable event names are coded into the software and maintained by developers



What ply can do

- kprobe – kernel function start
 - kprobe:do_sys_openat2
- kretprobe – kernel function return
 - kretprobe:do_sys_openat2
- tracepoint – kernel static tracepoints
 - tracepoint:sched/sched_wakeup



What ply can't do

- uprobe – user-level function start
 - uprobe:/bin/bash:readline
- uretprobe – user-level function return
 - uretprobe:/bin/bash:readline
- usdt – user-level static tracepoints
 - usdt:/usr/sbin/mysqld:mysql:query_start



Buildroot image for BeagleBone Black

- eBPF enabled in kernel
- /usr/sbin/ply and example scripts in /root
- git clone --recursive
<https://github.com/fvasquez/ply-bbb.git>
- cd ply-bbb
- make all
- output/images/sdcard.img
- root password is temppwd



Enabling eBPF in the kernel

- Start with 2021.08 release of Buildroot
- `beaglebone_defconfig` now uses a new beagleboard/linux 5.10 custom kernel
- Previous kernel version was 4.19
- `make beaglebone_defconfig`
- `make linux-configure`
- `make linux-menuconfig`



Common eBPF kernel config options

- At a minimum:
 - CONFIG_BPF=y
 - CONFIG_BPF_SYSCALL=y
- For BCC:
 - CONFIG_NET_CLS_BPF=m
 - CONFIG_NET_ACT_BPF=m
 - CONFIG_BPF_JIT=y
- From Linux 4.7 onward:
 - CONFIG_HAVE_EBPF_JIT=y
 - CONFIG_BPF_EVENTS=y



More eBPF kernel config options

- For pty:
 - CONFIG_KPROBES=y
 - CONFIG_TRACEPOINTS=y
 - CONFIG_FTRACE=y
 - CONFIG_DYNAMIC_FTRACE=y
- To stop 'trace_kprobe: Could not probe notrace function' errors:
 - CONFIG_KPROBE_EVENTS_ON_NOTRACE=y



Adding ply to a Buildroot image

- `make beaglebone_defconfig`
- `make menuconfig`
 - Target packages
 - Debugging, profiling and benchmark
 -  `ply`
- `make savedefconfig`
- `BR2_PACKAGE_PLY=y`



Target architectures

```
config BR2_PACKAGE_PLY_ARCH_SUPPORTS
    bool
    default y if BR2_aarch64
    default y if BR2_arm
    default y if BR2_powerpc
    default y if BR2_x86_64
```



ply needs

- Toolchain with dynamic library support
- Toolchain with kernel headers >= 4.14
- Kernel with eBPF enabled
- host_flex and host_bison at build time
- Run with CAP_SYS_ADMIN or as root
- debugfs mounted at /sys/kernel/debug



Command line options

Options:

- c COMMAND Run COMMAND in a shell,
exit upon completion.
- d Enable debug output.
- e Exit after compiling.
- h Print usage message and exit.
- k Keep going in face of trace
buffer overruns.
- S Show generated BPF.
- v Print version information.



One-liners

```
# Count vfs calls by executable & function
ply -c \
    "dd if=/dev/zero of=/dev/null bs=1 count=100" \
    'kprobe:vfs_* { @[comm, caller] = count(); }'

# Count syscalls systemwide by function
ply 'k:_se_sys_* { @syscalls[caller] = count(); }'
```



Count vfs calls by executable and function

```
ply: active
100+0 records in
100+0 records out
ply: deactivating

@:
{ dd          , vfs_statx }: 1
{ dd          , vfs_fstat }: 1
{ dd          , vfs_readlink }: 1
{ dd          , vfs_getattr_nosec }: 2
{ dd          , vfs_open }: 3
...
{ dropbear    , vfs_read }: 9
{ dropbear    , vfs_writev }: 9
{ ply         , vfs_read }: 19
{ ply         , vfs_open }: 25
{ dd          , vfs_read }: 101
{ dd          , vfs_write }: 108
```



Count syscalls systemwide by function

```
ply: active
^Cply: deactivating

@syscalls:
{ __se_sys_write }: 2
{ __se_sys_writev }: 2
{ sys_select }: 2
{ __se_sys_bpf }: 4
{ __se_sys_rt_sigaction }: 6
{ __se_sys_close }: 160
{ __se_sys_epoll_wait }: 160
{ __se_sys_perf_event_open }: 169
{ __se_sys_read }: 304
{ sys_brk }: 362
{ sys_ioctl }: 482
{ sys_clock_gettime32 }: 486
{ __se_sys_open }: 653
{ __se_sys_gettimeofday }: 1431
```



Probes

- ply programs consists of one or more probes which are analogous to awk's pattern-action statements

```
# single probe
<provider>:<probe-point(s)> /<predicate>/
{
    <statement>;
    ...
}
```



Providers

Probe kernel functions:

- 'kprobe' – Trace every time matching function(s) start
- 'kretprobe' – Trace every time matching functions(s) return

```
cat /proc/kallsyms
```

Probe kernel tracepoints:

- 'tracepoint' – Trace every time matching tracepoint(s) hit

```
ls /sys/kernel/debug/tracing/events
```



Built-ins

- 'comm' – name of running process's executable
- 'pid' – kernel thread group ID of running process
- 'kpid' – kernel pid of running process
- 'time' – nanoseconds elapsed since system boot
- 'printf("<format>", ...)' – Prints formatted output to `ply`'s standard out. Recognizes '%v' which will dump the value according to the inferred type's default (i.e. how 'print' would print it).



Provider-specific

'kprobe' and 'kretprobe':

- 'stack' – kernel stack trace to current location

'kprobe' only:

- 'caller' – name of function that triggered probe
- 'arg0', 'arg1' ... 'arg<N>' – argument values passed to function that triggered probe

'kretprobe' only:

- 'retval' – return value of probed function



Maps

- Store data in a map by assigning a value to a key

```
<name>[<exprs>] = <expr>
```

```
rx[arg0] = time
```

- Extract data to end user and carry data between probes (global scope)

```
<name>[<exprs>]  
time - rx[arg0]
```



Aggregations

- Maps that can be assigned the result of aggregation functions (i.e. count and quantize)
- Start with '@'

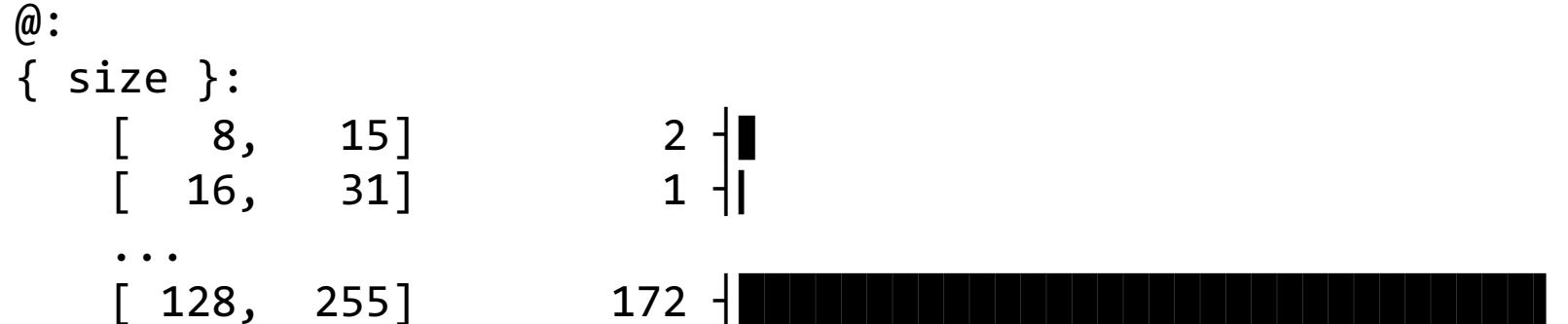
```
# Bump a counter  
@<name>[<exprs>] = count()
```

```
# Store the distribution of an expression  
@<name>[<exprs>] = quantize(<scalar-expr>)
```



Read size distribution

```
# ply 'kretprobe:sys_read { @["size"] = quantize(retval); }'  
ply: active  
^Cply: deactivating
```



opensnoop.ply

```
#!/usr/sbin/ply

kprobe:do_sys_openat2
{
    path[kpid] = str(arg1);
}

kretprobe:do_sys_openat2
{
    printf("%v %v %v :%d\n", pid, comm, path[kpid], retval);
}
```



pid comm path[kpid] :retval

```
# ./opensnoop.ply
ply: active
284 redis-server      /etc/TZ          :-2
284 redis-server      /etc/localtime   :-2
284 redis-server      /proc/284/stat    :8
284 redis-server      /etc/TZ          :-2
284 redis-server      /etc/localtime   :-2
284 redis-server      /proc/284/stat    :8
284 redis-server      /etc/TZ          :-2
284 redis-server      /etc/localtime   :-2
284 redis-server      /proc/284/stat    :8
^Cply: deactivating
```



execsnoop.ply

```
#!/usr/sbin/ply
```

```
kprobe:sys_execve {  
    execs[kpid] = str(arg0, 48);  
}
```

```
kretprobe:sys_execve {  
    printf("(%4u) %v %3ld\n", uid, execs[kpid], retval);  
}
```



(uid) execs[kpid] retval

```
# ./execsnoop.ply &
# ply: active

# /etc/init.d/S50redis stop
( 0) /etc/init.d/S50redis                               0
Stopping redis: ( 0) /usr/bin/redis-cli                0
OK

# /etc/init.d/S50redis start
( 0) /etc/init.d/S50redis                               0
Starting redis: ( 0) /sbin/start-stop-daemon          0
OK

# (1002) /usr/bin/redis-server                         0
```



```
execs[kpid] = str(arg0, 48)
```

```
# fg %1
./execsnoop.ply
^Cply: deactivating
```

```
execs:
{ 270 }: /etc/init.d/S50redis
{ 272 }: /etc/init.d/S50redis
{ 273 }: /sbin/start-stop-daemon
{ 271 }: /usr/bin/redis-cli
{ 275 }: /usr/bin/redis-server
```



tcp-send-recv.ply

```
#!/usr/sbin/ply

kprobe:tcp_sendmsg
{
    @[comm, "send"] = count();
}

kprobe:tcp_recvmsg
{
    @[comm, "recv"] = count();
}
```



Count TCP I/O by executable and direction

```
# ./tcp-send-recv.ply &
# ply: active

# redis-cli --latency
min: 0, max: 5, avg: 1.20 (1025 samples)^C
# fg %1
./tcp-send-recv.ply
^Cply: deactivating

@:
{ dropbear      , recv      }: 32
{ redis-cli     , recv      }: 1025
{ redis-cli     , send      }: 1025
{ redis-server   , send      }: 1025
{ redis-server   , recv      }: 1026
{ dropbear      , send      }: 1041
```



heap-allocs.ply

```
#!/usr/sbin/ply

# heap allocation size distribution
kprobe:sys_brk
{
    @["alloc size"] = quantize(arg0 - heaps[comm, kpid]);
    heaps[comm, kpid] = arg0;
}
```



Run LRU simulator using 100k keys

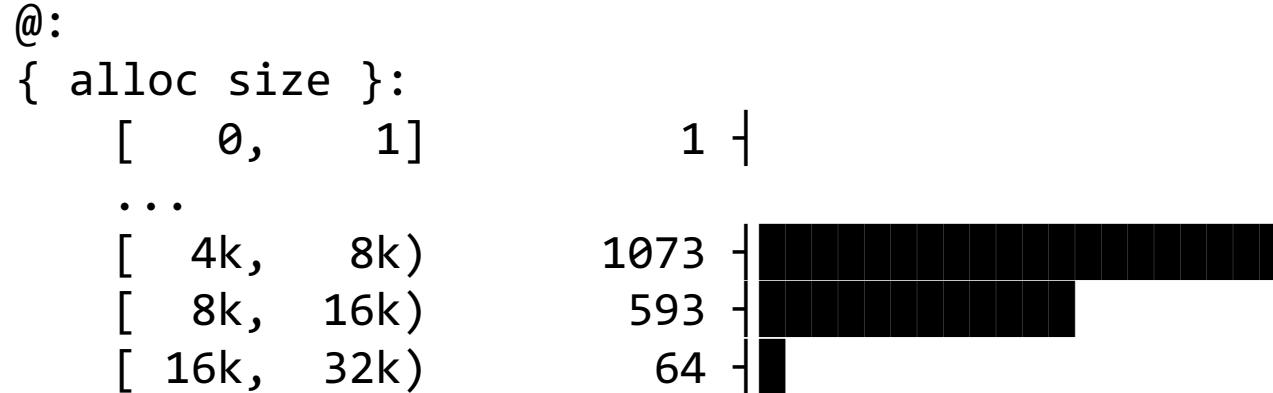
```
# redis-cli flushall
OK
# ./heap-allocs.ply &
# ply: active

# redis-cli --lru-test 100000
7000 Gets/sec | Hits: 880 (12.57%) | Misses: 6120 (87.43%)
7000 Gets/sec | Hits: 2242 (32.03%) | Misses: 4758 (67.97%)
7000 Gets/sec | Hits: 3260 (46.57%) | Misses: 3740 (53.43%)
6750 Gets/sec | Hits: 3797 (56.25%) | Misses: 2953 (43.75%)
7000 Gets/sec | Hits: 4441 (63.44%) | Misses: 2559 (36.56%)
7000 Gets/sec | Hits: 4826 (68.94%) | Misses: 2174 (31.06%)
7000 Gets/sec | Hits: 5154 (73.63%) | Misses: 1846 (26.37%)
7000 Gets/sec | Hits: 5426 (77.51%) | Misses: 1574 (22.49%)
7000 Gets/sec | Hits: 5583 (79.76%) | Misses: 1417 (20.24%)
^C
```



Heap allocation size distribution

```
# fg %1  
./heap-allocs.ply  
^Cply: deactivating
```



Buy MELP3 at Amazon
<https://packt.live/3tiDDrA>



fvasquez



@st8l3ss



st8l3ss#9518

