BPFd

BPFd: Powerful Linux Tracing for Remote targets using eBPF

Joel Fernandes <joelaf@google.com>
LinuxInternals.org
IRC: joel_ on OFTC

Google

About me:

Kernel team
- scheduler
- tracing

Demos of working tools on Android/Hikey960

filetop: Displays File I/O summary every 5 seconds

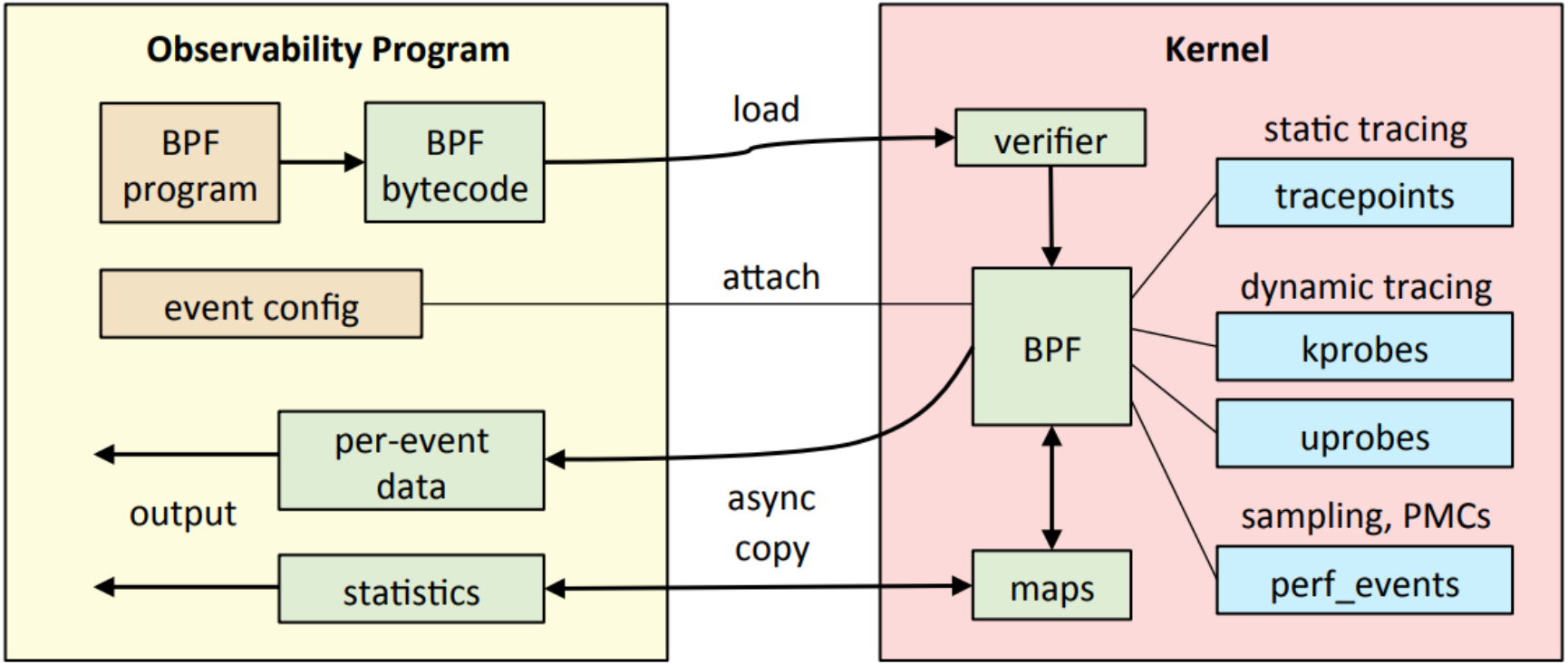
```
# ./tools/filetop.py 5

This tells the tool to monitor file I/O every 5 seconds. While filetop
was running, start the stock email app in Android:
Example, opening contacts app, and create a contact.

TID    COMM             READS  WRITES R_Kb    W_Kb    T FILE
6726   Binder:6152_8    29     0      112     0       R contacts2.db
6726   Binder:6152_8    26     44     104     88      R contacts2.db-wal
2107   servicemanager   16     0      63      0       R current
2107   servicemanager   14     0      55      0       R current
6166   Binder:6152_2    9      0      36      0       R contacts2.db-wal
6166   Binder:6152_2    8      0      32      0       R contacts2.db
5747   Profile Saver    3      0      16      0       R primary.prof
6479   Binder:6152_5    3      0      12      0       R contacts2.db
```

Signals of interesting things in the kernel:

- static trace points (ftrace events)
- dynamic trace points (kprobe)
- userspace dynamic trace points (uprobes)
- userspace static tracepoints (usdt+uprobes)
- perf events – PMC counters (cycles, cache misses)
- perf profiling/sampling.

BPF for Tracing, Internals

BPF lets you attach and observe them

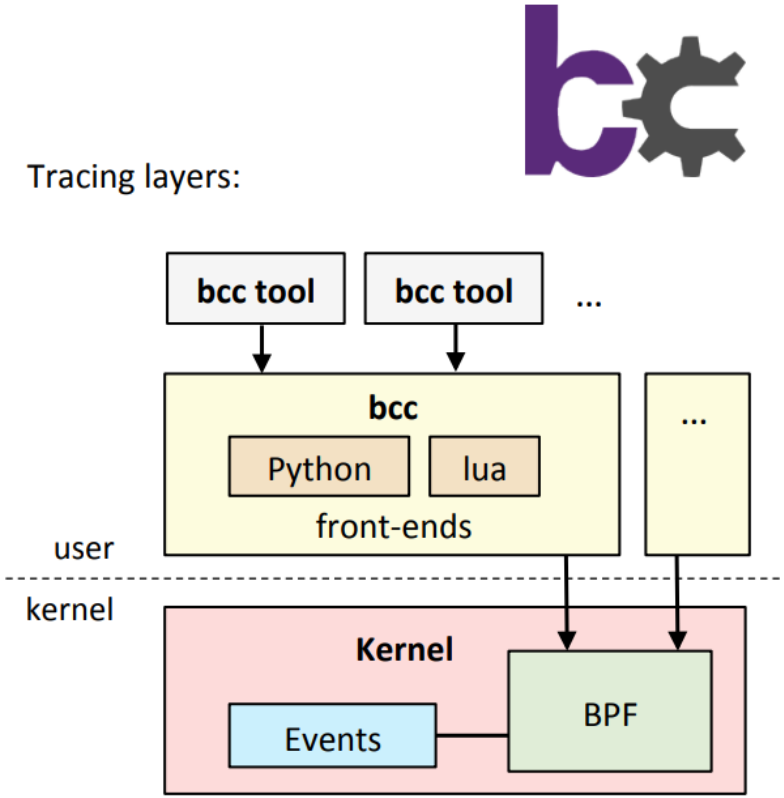


Enhanced BPF is also now used for SDNs, DDOS mitigation, intrusion detection, container security, ...

Credit: Brendan Gregg

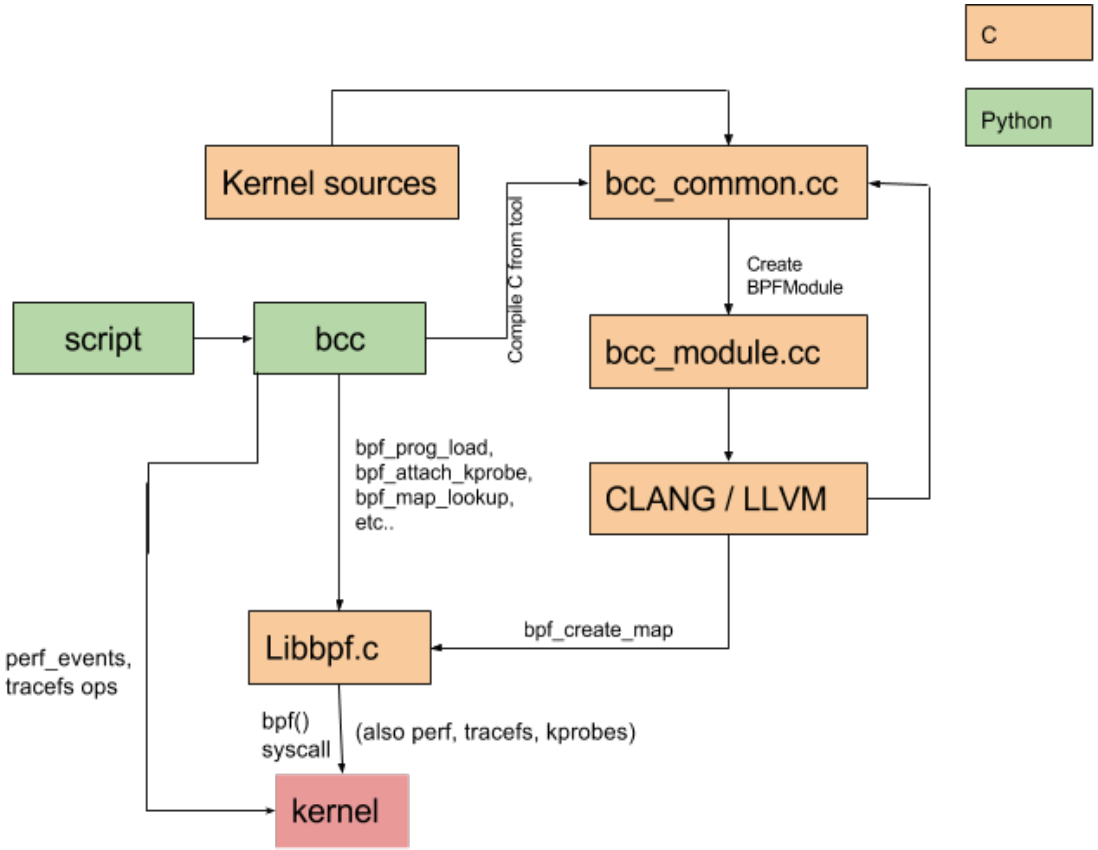
BCC Sales pitch

Summarizing.



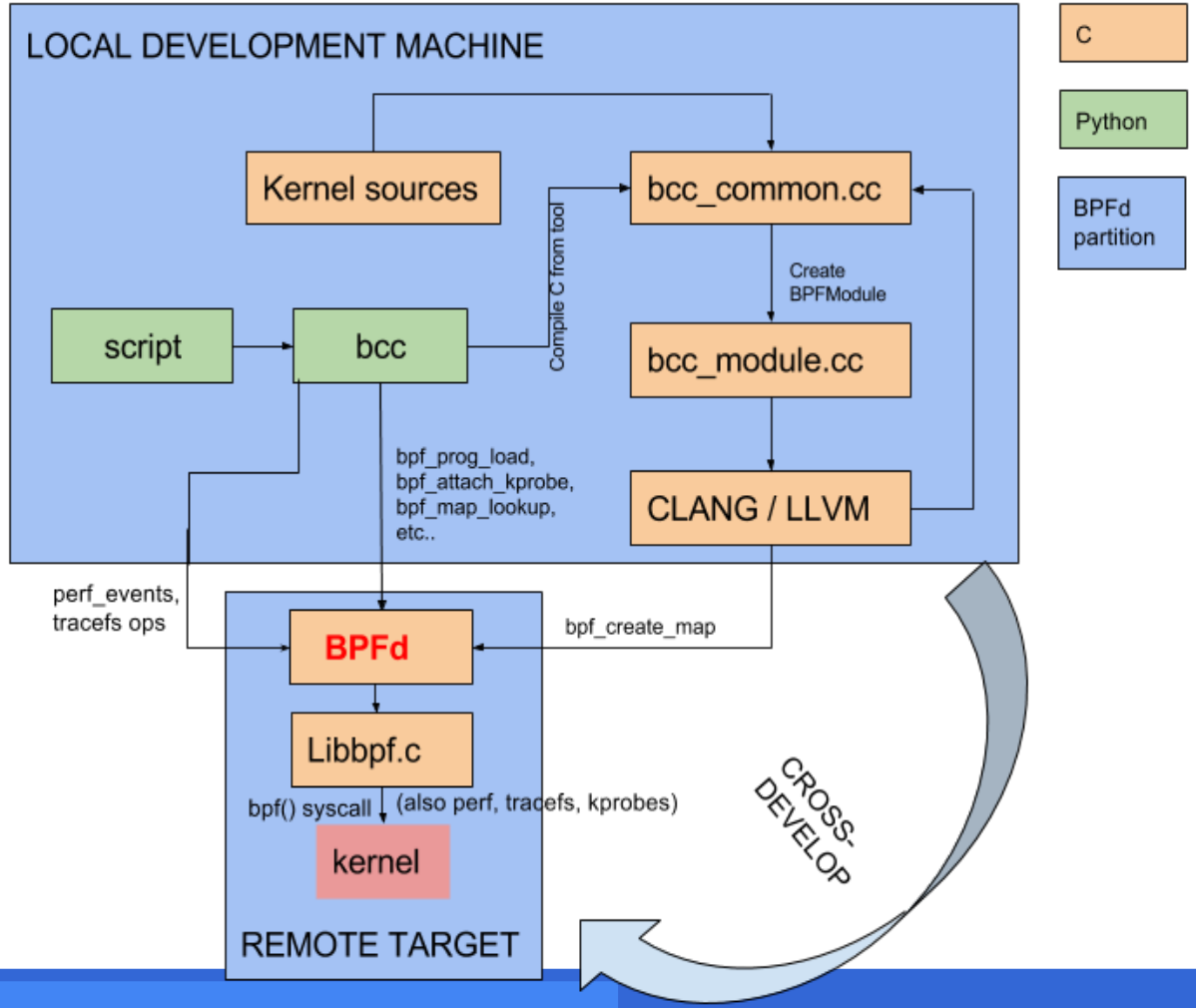
Credit: Brendan Gregg

Traditional BCC:



Can’t run on different system or different Architecture! Big problem!

BCC for remote:



BPFd: Why we want to do it?

- ➔ Pain downloading and sync'ing kernel sources to the remote target
- ➔ Need for cross-compiled clang, llvm, python stack on target
- ➔ Fits well with cross development workflow for Embedded developers (Build on host, run on target)
- ➔ Availability of symbols, dwarf debug info, etc on host if needed
- ➔ Host machines usually much powerful and better for processing than battery powered device.

Status: What works

What works in Upstream:

- BCC fixed for ARM64 platforms (Added October '17)
- Support to Compile for any architecture dynamically (Jan '18)
- BCC Support to compile eBPF on custom kernel tree (Jan '18)
- Preliminary support for BCC communicating to remote targets (Jan '18)
  - Project started for that: BPFd (https://lwn.net/Articles/744522/)
  - Refactoring BCC to make it easier to add remote support merged.

What works down stream:

- All of the above.
- BCC remote support to talk to remote targets
  - Kprobes, tracepoints
  - Several tools: biosnoop, filetop, hardirq

Demos of working tools on Android/Hikey960

hardirq: Total time spent in hard interrupt handlers
Example. Start and minimize an app a lot, watch the mali interrupts total time:

```
# ./tools/hardirqs.py 10

 Tracing hard irq event time... Hit Ctrl-C to end.
HARDIRQ                    TOTAL_usecs
wl18xx                             181
ufshcd                             243
dw-mci                             409
hisi-asp-dma                      2671
mailbox-2                         2842
timer                             9978
xhci-hcd:usb1                    12468
kirin                            13720
e82c0000.mali                    60635
```

Demos of working tools on Android/Hikey960

biotop: Summary of processes and Block I/O

```
PID    COMM             D MAJ MIN DISK       I/O  Kbytes  AVGms
4524   droid.gallery3d  R 8   48  ?           33   1744   0.51
2135   jbd2/sdd13-8     W 8   48  ?           15    356   0.32
4313   kworker/u16:4    W 8   48  ?           26    232   1.61
4529   Jit thread pool  R 8   48  ?            4    184   0.27
2135   jbd2/sdd13-8     R 8   48  ?            7     68   2.19
2459   LazyTaskWriterT  W 8   48  ?            3     12   1.77
```

biosnoop: Trace like view of all Block I/O operations and their latency

```
  TIME(s)        COMM           PID    DISK    T  SECTOR    BYTES   LAT(ms)
  0.000000000    jbd2/sdd13-8   2135   sdd     W  37414248  28672      1.90
  0.001563000    jbd2/sdd13-8   2135   sdd     W  37414304  4096       0.43
  0.003715000    jbd2/sdd13-8   2135   sdd     R  20648736  4096       1.94
  5.119298000    kworker/u16:1  3848   sdd     W  11968512  8192       1.72
  5.119421000    kworker/u16:1  3848   sdd     W  20357128  4096       1.80
  5.448831000    SettingsProvid 2415   sdd     W  20648752  8192       1.70
```

Demos of working tools on Android/Hikey960

cachestat: Page Cache Reads

Hits

```
# while [ 1 ]; do cat 1; sleep 1; done > /dev/null
```

```
   TOTAL   MISSES     HITS  DIRTIES   BUFFERS_MB  CACHED_MB
       0        0        0        0          208       1794
       0        0        0        0          208       1794
   27045        0    27045        0          208       1794
   55603        0    55603        0          208       1794
   56313        0    56313        0          208       1794
   33567        0    33567        0          208       1794
   56313        0    56313        0          208       1794
   33567        0    33567        0          208       1794
```

Demos of working tools on Android/Hikey960

cachestat: Page Cache Reads

Misses

```
# while [ 1 ]; do echo 1 > /proc/sys/vm/drop_caches; cat 1; sleep 1; done > /dev/null
```

```
   TOTAL   MISSES     HITS  DIRTIES   BUFFERS_MB  CACHED_MB

     767        0      767        0          208       1794
   54657    51727     2930        0          208       1794
   55616    51894     3722        0          208       1794
   28526    25949     2577        0          208       1794
   52006    48992     3014        0          208       1794
   55602    51864     3738        0          208       1794
```

Demos of working tools: Trace Multitool

Usecase: Tracing Kprobes from trace multitool

```
long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode) { .. }
```

```
# ./tools/trace.py 'do_sys_open "%s", arg2' -T

TIME     PID     TID     COMM            FUNC             -
19:45:44 2220    2250    storaged        do_sys_open      /sys/block/sda/stat
19:45:44 2220    2250    storaged        do_sys_open      /sys/block/sda/stat
19:45:48 2132    2132    servicemanager  do_sys_open      /proc/4113/attr/current
19:45:49 2352    2437    DeviceStorageMo do_sys_open      /system/framework/arm/boot.art
19:45:49 2352    2437    DeviceStorageMo do_sys_open      /data/dalvik-cache/arm/system@framework@boot.art
19:45:49 2352    2437    DeviceStorageMo do_sys_open      /system/framework/arm64/boot.art
19:45:49 2352    2437    DeviceStorageMo do_sys_open      ../system@framework@boot.art
19:45:55 2132    2132    servicemanager  do_sys_open      /proc/2480/attr/current
19:45:55 2132    2132    servicemanager  do_sys_open      /proc/2480/attr/current
```

Demos of working tools: Trace Multitool

Tracing Kernel tracepoints! (Can also do user tracepoints, once Android gets USDT).

```
./tools/trace.py  't:block:block_rq_complete "sectors=%d", args→nr_sector'

PID     TID     COMM            FUNC             -
0       0       swapper/0       block_rq_complete sectors=64
0       0       swapper/0       block_rq_complete sectors=0
0       0       swapper/0       block_rq_complete sectors=8
0       0       swapper/0       block_rq_complete sectors=0
0       0       swapper/0       block_rq_complete sectors=80
0       0       swapper/0       block_rq_complete sectors=0
```

Running a BPF program during a profiling event

```
# ./runqlen.py
Sampling run queue length... Hit Ctrl-C to end.
^C
     runqlen       : count     distribution
        0          : 1068     |****************************************|
        1          : 642      |************************                |
        2          : 369      |*************                           |
        3          : 183      |******                                  |
        4          : 104      |***                                     |
        5          : 42       |*                                       |
        6          : 13       |                                        |
        7          : 2        |                                        |
        8          : 1        |                                        |
```

Status: What doesn't work (yet) & TODO

Boring issues that have a path to being fixed:

- Uprobes (Userspace dynamic tracing – almost working)
- USDT (Userspace statically defined tracing)
- Symbol lookup failures causing:
  - Stack symbol deref issues – both kernel and user
  - Tools that depend on sym info fail (like cachestat)
- Support for tools that need to run locally (Some tools read the process table for example)
- Kernel version issues
- Full List: https://github.com/joelagnel/bpfd/issues

More Interesting issues

- BPFd activity causes tracer side effects
  - Solution: blacklist BPFd process from being traced?
- Some tools like trace multi-tool can generate a lot of output.
- Perf polling cannot be interrupted by CTRL^C
- Implementing a remote logging mechanism – currently logging is turned off

Resources

- BPFd project: https://github.com/joelagnel/bpfd
- LWN article: https://lwn.net/Articles/744522/
- Brendan Gregg's eBPF page: http://brendangregg.com/perf.html#eBPF

Thanks

- Thanks Jazel Canseco for reviewing presentation.
- Brendan Gregg, Alexei Staravoitov and Sasha Goldstein for encouragement.
- Android kernel team for encouragement and ideas.

Questions?