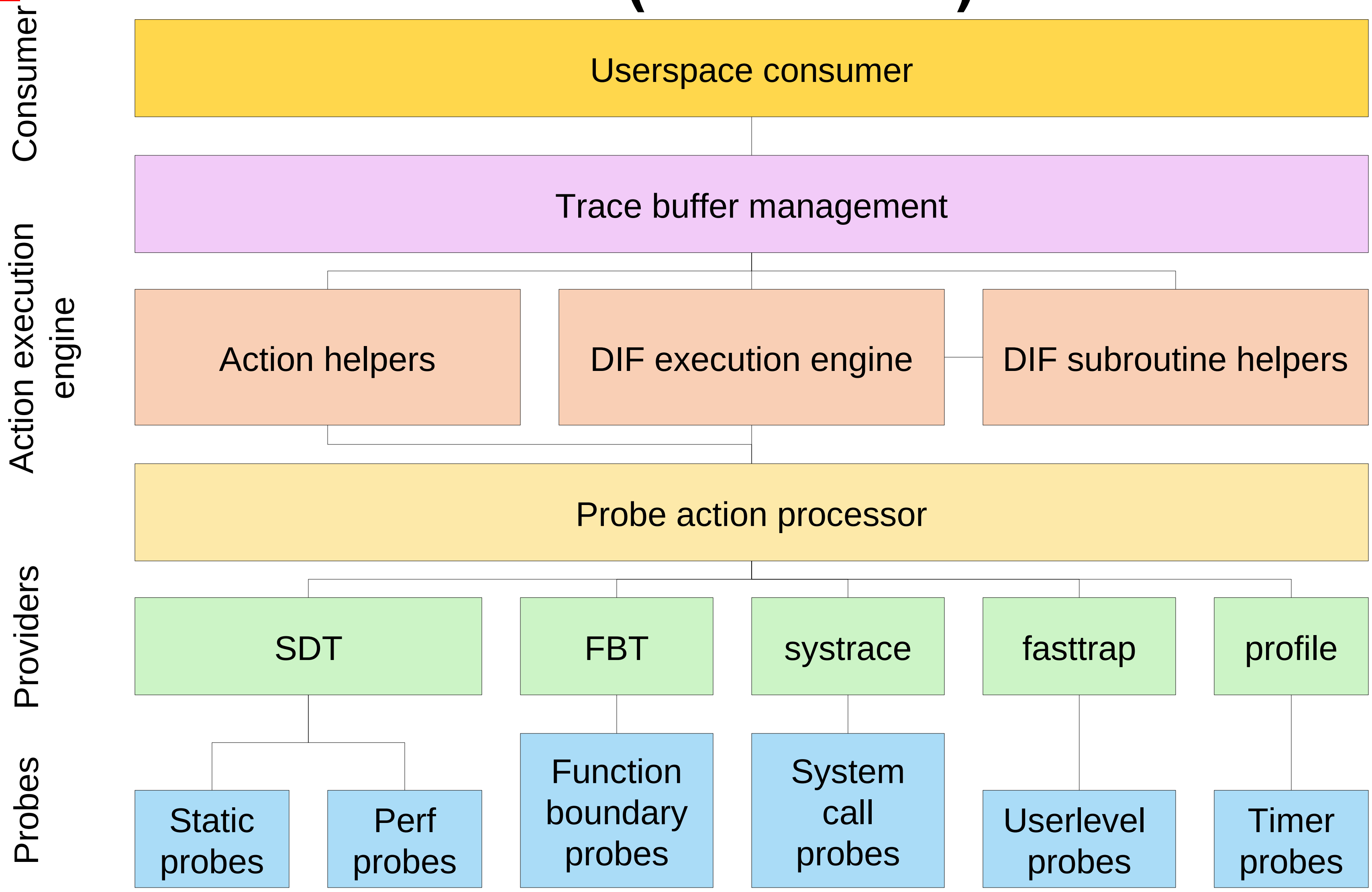
eBPF as execution engine for DTrace

dr. Kris Van Hees
Consulting Engineer, Languages and Tools
Linux Engineering

DTrace on Linux (w/o eBPF)



DTrace

- Userspace:
  - Probe context: registers, arguments, …
  - Task context: pid, ppid, uid/gid, euid/egid, comm, …
  - Consumer context: buffers, ...
- Kernel:
  - Statically Defined Tracing (SDT) probes
  - Low-level probe firing mechanisms
  - DTrace specific task management
  - Expose DTrace kernel features to DTrace kernel modules
- Kernel modules:
  - Core DTrace module: API to providers, probe action execution, buffer management
  - Provider modules: expose probes to DTrace core, implement generic probe API, probe firing mechanism

eBPF

- Based on the Berkeley Packet Filter (BPF) project
- Extended to be a bytecode-based execution engine
- Designed to be safe and fast
- Designed to support easy Just-In-Time compilation

- Originally used for network filters
- Now you can attach BPF programs to various other things:
    - kprobes / uprobes
    - tracepoints
    - perf events
    - ...

Tracing facilities in the Linux kernel

- SDT: tracepoints
- FBT: kprobe / kretprobe
- Pid: uprobe / uretprobe
- Profile: software timer perf events
- Syscall: tracepoints (sys_enter_*, sys_exit_*)

- All are exposed through */sys/kernel/debug/tracing/events*
- All are presented as tracing events, and eBPF programs can be attached to all of them
- All tracing probes can use the perf_event_output helper to write output to a perf_event output ring buffer

Tracing with eBPF

- Create a kprobe/uprobe, or “open” a perf event
- Load a eBPF program (using the bpf() system call)
- Attach the eBPF program to the perf event
- [Enable the probe]

- eBPF program writes output using bpf_perf_event_output()
- Userspace reads from the perf_event ring-buffer when data is available

- eBPF programs are usually compiled using Clang/LLVM

- Pretty straightforward… or so it seems

Complications

- Each BPF program consumes *n* pages (*n* >= 1)
- Probe specific program types, with probe-specific context
- Each program type has its own list of accessible helpers
- Not all task data can be obtained with a helper (e.g. ppid)
- BPF does not allow dereferencing pointers
- Limited output options:

  bpf_trace_printk() - add message to trace buffer

  bpf_perf_event_output() - add event sample to ring buffer

DTrace

- D programs (DIF code) execute in a DTrace context
- All probe types trigger execution in that same context
- DTrace generates efficient output (no need for meta-data)

- Big differences between eBPF and DTrace:
  - eBPF: probe executes BPF program
  - DTrace: probe triggers execution of DIF code fragments

  - eBPF: output encapsulated in perf_event sample data
  - DTrace: raw data

  - Linux probes/events do not map well to the standard DTrace probe naming: *provider:module:function:name*

ORACLE

DTrace workflow (before eBPF)

- D scripts are a collection of clauses each tied to one or more probes
- Each clause is a sequence of actions (some generate data, some manipulate variables, some perform more complex functions)
- Each action usually has some D expression associated with it, compiled into Dtrace Intermediate Format (DIF) code
- When a probe fires, the execution engine loops through all clauses associated with it
- For every clause, the execution engine loops through all actions that are part of it
- For every action, if there is a D expression associated with it, it is executed by the DIF emulator
- … it must have been a good idea at the time...

DTrace based on eBPF (1st attempt)

- Redesign of DTrace based on eBPF and kernel facilities
- Identified some “shortcomings”
- Proposed patches to eBPF and other kernel components to support a more tracing-centric general design

- Patches were rejected because kernel developers did not believe they were necessary

DTrace based on eBPF (2nd attempt)

- New philosophy: Let's assume we can implement DTrace without any kernel modifications
- Assume that we can do this without impacting the performance and stability we've grown accustomed to
- Perform accuracy, stability and performance tests
- Use results to either confirm that kernel modification are not needed, or to provide evidence that modifications to the kernel are needed

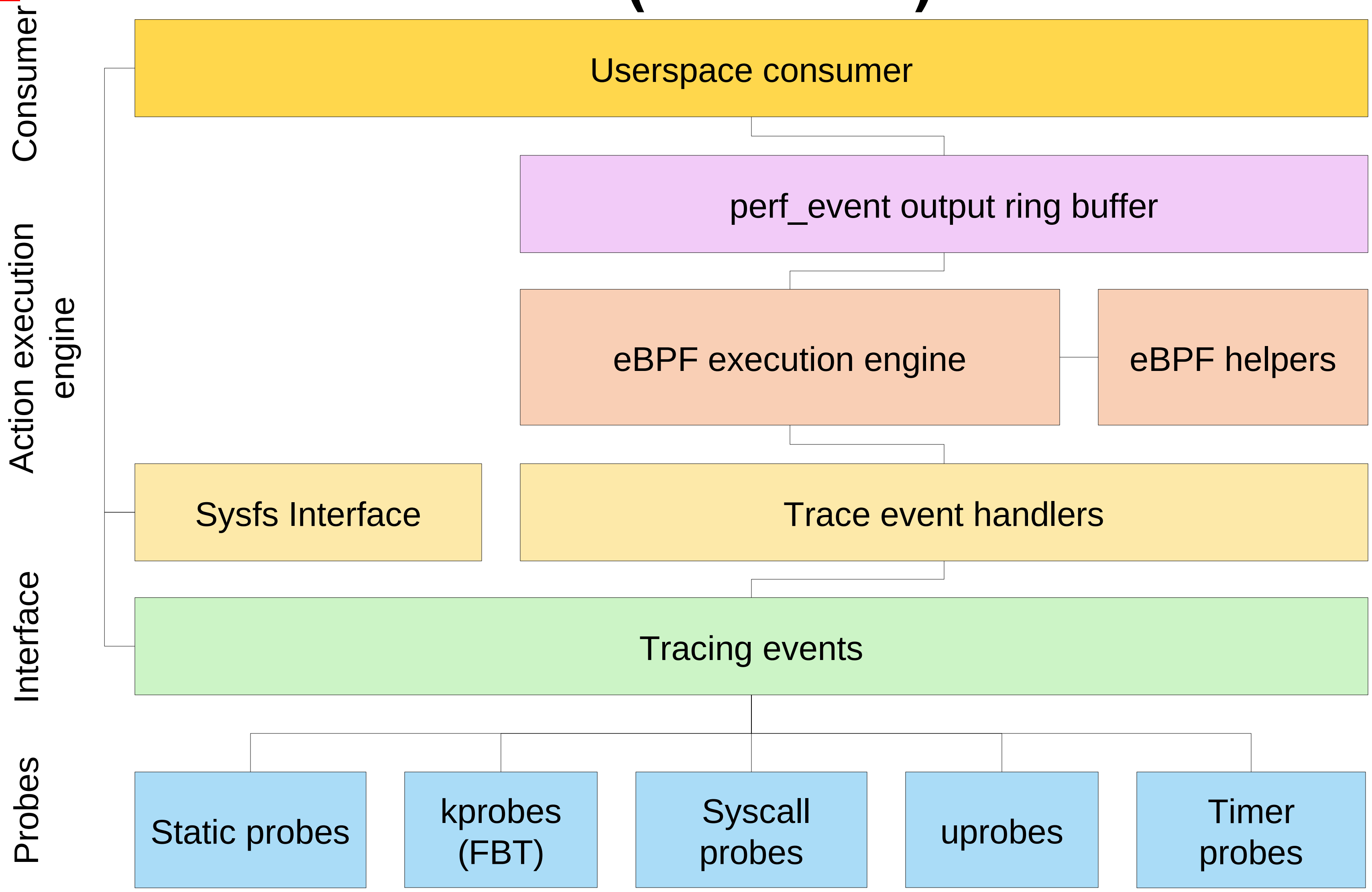
- Still in progress...

Before we go on… Why?

- DTrace has been around for quite a long time
  - Quite a few people are familiar with it
  - Its feature set has been very well documented
  - It has proven to be quite good at what it does
  - It has been ported to multiple OSes
- DTrace provides a powerful programmable tracing system
  - Easy to do very basic tracing
  - Powerful enough to support use cases that involve complex combinations of probes
  - Stable enough to do long-term tracing (even always-on)
- People want it.

- DTrace can break through some of the limitations imposed by its original design without changing how it works

DTrace on Linux (w/ eBPF)



DTrace v2 based on eBPF

- Generate an eBPF program for each D script clause
- Generate an eBPF trampoline program for each probe
    - Set up an ECB structure to capture DTrace state
    - Call the eBPF program associated with the probe
- Attach the trampoline to the probe
- Provide eBPF functions to implement specific actions

- All functionality is moved into userspace
- It is unlikely that this approach scales well with large numbers of probes

- Advantage: problems we find benefit other tracing projects!

DTrace v2 based on eBPF (cont.)

- DTrace has its own compiler to eBPF (D to eBPF)
- Full control over what data to collect, how to collect it, and how to prepare it for post-processing

- Very big paradigm shift for Dtrace:
  - Before: DTrace was kernel based with a userspace front
  - Now: DTrace becomes a user of existing facilities
  - Advantage: We can actually contribute to the overall Linux framework
  - Advantage: We don’t have to maintain everything ourselves (← My favourite!)

DTrace v2: Pending contributions

- Compact C Type Format (CTF) data
  - Emphasis on “compact”
  - Necessary for function arguments, typed access to kernel data
- /proc/kallmodsyms
  - Similar to /proc/kallsyms
  - Add symbol size information
  - Add module name info (even for builtin modules)
  - Needed to provide stable probe naming regardless of whether modules are compiled in or loadable

Unsolved mysteries...

- Is the existing set of probes in Linux sufficient for what DTrace has traditionally provided (especially documented probes that are expected to be available with DTrace).
- Should we use custom trampoline eBPF programs that “translate” existing probes into probes we need?
- What is the best way to contribute new probes to the kernel (not specific to DTrace).
- Can we support tracing using thousands of probes?
- How to get past eBPF limits (e.g. a probe can only have 64 eBPF programs attached to it)

- And, and, and, ...

Where to find things...

- Compiler support for eBPF added to gcc *[JM]*
  - Sent to gcc-patches last week
- Toolchain support for eBPF added to binutils *[JM]*
- Compact C Type Format support added to binutils *[NA]*
  - https://sourceware.org/git/binutils-gdb.git (master)
- Libbpf to interact with eBPF and perv events in the kernel
  - Included in DTrace (modified version of libbpf from the kernel source tree)
  - Will be obsolete in the near future (we only use a very small portion of the functionality it provides)
- DTrace (very much a work in progress)
  - https://github.com/oracle/dtrace-utils (2.0-branch)

JM = José Marchesi, NA = Nick Alcock