DTrace on Linux

Kris Van Hees
Languages and Tools
Linux Engineering
<kris.van.hees@oracle.com>
Overview

- (Very) short history
- (Very) short DTrace overview
- DTrace using BPF, etc
- Significant implementation details
- Unanswered questions
(Very) short history

- DTrace on Linux started in 2010
- First version in Oct 2011
- Under active development every since
- Redesign without big kernel patches
  - Planning since mid-2018
  - Coding started July 2019
(Very) short DTrace overview

- Two components:
  - Kernel space producer (~45K lines)
    - Core kernel support functions
    - Core kernel probes
    - DTrace core and provider modules
  - Userspace consumer (~55K lines)
    - Userspace library and front-end
DTrace using BPF, etc

- Kernel provides probing mechanisms
- BPF gives us an execution engine
- BPF programs attach to probes
- Output written to perf_event ring buffer
Not *that* easy!

**BPF**
- Probe specific program types
- Probe specific context
- One program per probe

**DTrace**
- Single program type
- Consistent probe context
- Many clauses per probe
Design philosophy

- Assume we can do everything in userspace
- Assume this will not impact performance and stability
- Keep dreaming
Design philosophy (revised)

- Assume we can do everything in userspace
- Assume this will not impact performance and stability
- Re-implement DTrace in userspace
- Perform accuracy, performance, and stability tests
- Evaluate findings:
  - Confirm kernel patches are not needed, or
  - Kernel patches are needed (and we can show why)
Implementation details

- Each D clause is compiled into a BPF function `dt_func(dt_dctx_t *dctx)`
- BPF trampoline program generated for each probe that is being enabled
- Trampoline calls the BPF functions for the probe clauses
- Completely different from what DTrace used to do
- Much more elegant… but…
Implementation details

- Compile entire clauses instead of actions
- Compiler re-targeted to BPF
- Disassembler re-targeted to BPF
- Added a linker to construct programs
- Implement memory management for local, global, and TLS variables
- BPF support functions (compiled with gcc)
Unanswered questions

• Impact of lack of code sharing
• Pointer value identification
  – Pointer to BPF memory (stack, map value) → direct deref possible
  – Pointer to kernel memory → bpf_probe_read()
• Dynamic variables
• ERROR probes (esp. arguments)
• Standard DTrace SDT probes
• String manipulation functions
• Scalability (what if I need to probe 1000s of probes)
Where to find it?

- Source code:  
  
  http://github.com/oracle/dtrace-utils/tree/2.0-branch-dev

- Mailing list:  
  
  dtrace-devel@oss.oracle.com
Why?

- People want it!
  - DTrace has been around for a long time
  - Well documented feature set
  - Available on multiple operating systems

- Powerful programmable tracing system
  - Easy to do very basic tracing
  - Powerful enough for complex tracing across many probes
  - Stable enough for long-term tracing (incl. Always-on tracing)

- Easier to develop new features for it