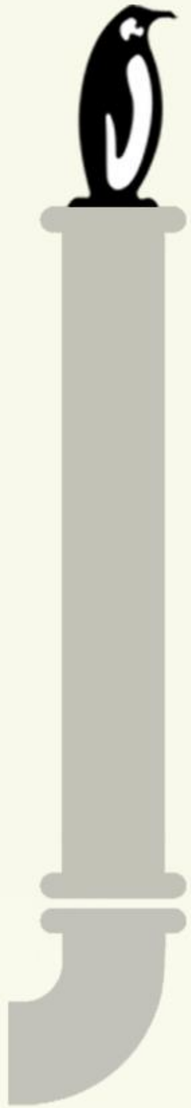


Linux
Plumbers
Conference | Richmond, VA | Nov. 13-15, 2023

DTrace and eBPF: new challenges

dr. Kris Van Hees
kris.van.hees@oracle.com
Languages and Tools
Linux Engineering
Oracle

Overview

- VERY brief overview of DTrace on Linux
- Status update
- Challenges:
    - Product status
    - Verifier limitations
    - Consistent naming of symbol in modules
    - BTF type data limitations

What is DTrace?

DTrace was first released in 2004 as an answer to the question:
**“If a complicated [production] system is doing something unexpected,
how do you determine what it is doing and why,
without taking the system down?”**

Is it the kernel, or user space, or both, or one triggering the other, …

We need a **Programmable dynamic tracing tool**

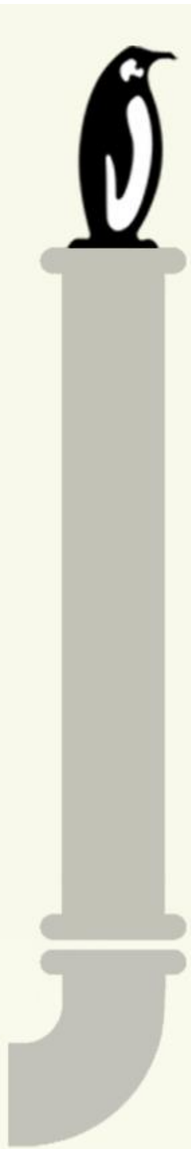
Programmable dynamic tracing tool

- Programmable:
  - C-style scripting language
  - Clauses associated with probes (n-to-n relation)
  - Conditional clauses with predicates
  - Global, local, and TLS variables
  - …
- Dynamic:
  - Script execution can adapt to trace data
  - …

Status update

- Near feature parity with earlier version

- Support for alternatives to kernel patches that are not upstream (yet)
  - no more waitfd()
  - kallsyms is sufficient (but imposes some limitations)
  - BTF is sufficient (but imposes some limitations)



Status update (summary)

DTrace works on systems with standard kernels!

(While the very vast majority of functionality works with stock kernels, some features will have some limitations, such as the need to cast kernel variable values due to lack of type data for variables, and the inability to support consistent naming of kernel symbols in modules due to the lack of module name information for symbols in built-in kernel modules.)

Challenges: Product status

- DTrace is a product
- ,,, so it needs to work on a range of supported kernels

- But the set of helpers differs between kernel versions
- But the BPF verifier differs between kernel versions
- But kernel changes affect target areas for tracing
- …

- These are unavoidable complications – but worth it.

Challenges: Verifier limitations

- BPF verifier may reject valid programs if it cannot guarantee safety
  - DTrace compiler favors verifiable code over better code
- BPF verifier has limited state tracking
  - Semantic relations between variables pose problems
- BPF verifier limits evaluation to 1 million instructions
  - Seems like a lot
  - But it is easy to reach that limit
  - Branches, function calls, and loops make it even easier

Challenges: Verifier limitations

```
for (i = j = 0; i < 10; i++, j++)
        arr[j] = 0;
```

The BPF verifier cannot verify bounds on the `arr[j]` access because it does not know that `i` and `j` are related.

Challenges: Tracing sub-system behaviour

- Not all that is listed can be probed:
  - Some functions listed in available_filter_functions cannot be probed
- Reasons for probe creation failures can be quite obscure (to a user)
  - Even more fun when probe creation works but attaching BPF fails
- Creating (and removing) kprobes/uprobes can be quite slow
  - No big deal when only creating a few
  - Gets slower as there are more registered kprobes/uprobes

Challenges: Symbol naming in modules

- We want to refer to a symbol by a <module name, symbol name> pair
- Makes it possible to perform symbol lookup in a specific module
- But what if the module is not loadable, but built in?

- Trace scripts that refer to a module symbol should work whether the module is loadable or built in
- But built in modules are simply part of the kernel.

Symbol naming in modules: possible solution

At kernel build time:

- Parse linker map data to determine what address ranges cover CUs that belong to one or more built-in modules
- Generate a list of these address ranges with the associated module name(s)

At runtime:

- Use address range data together with kallsyms to know whether an address falls within a built-in module

Then <module name, symbol name> works for built-in and loadable modules!

Challenges: Lack of type data for kernel vars

- BTF does not store type data for kernel variables
- (Actually, it has type data for per-cpu variables **only**.)

- Tools need to explicitly cast kernel variables to their proper type to access their value.
- This makes tracing scripts a lot less user friendly
- And the actual types **are** usually in BTF anyway
- But not associated with a particular variable

- Having BTF type data for kernel variables would help all tracers

Thank you!

https://github.com/oracle/dtrace-utils/tree/devel

dtrace-devel@oss.oracle.com