



Probe events with typecast using BTF

Define probes/triggers in a more C-like way

Masami Hiramatsu

Google / Linux Kernel Tracing & Bootconfig Maintainer
<mhiramat>

Linux Plumbers Conference 2026 tracing MC



LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

BTF Usages in Kernel Tracing

BPF Type Format (BTF) has fundamentally evolved kernel tracing by providing rich, compiler-generated type information.

By capturing the precise memory layout of C types, structs, and function signatures directly within the kernel image, BTF enables **safe, accurate context reflection**. This completely removes the legacy manual guesswork and physical offsets from kernel memory dereferencing.

Today, BTF is utilized extensively across high-performance **BPF tracing hooks**, dynamic kernel probes (including **kprobes**, **fprobes**, and **eprobes**), as well as standard trace events configured inside the core **kernel/trace/** subsystem.

History of BTF in Tracing

- **v5.5:** Raw Tracepoint Contexts
- **v5.7 / v5.8:** PROG_TRACING (BPF Trampolines, tp_btf)
- **v5.10:** bpf_seq_printf_btf & bpf_snprintf_btf
- **v6.5:** Function Parameter Access & Typed \$retval
- **v6.6:** Struct Field Dereferencing & Module Support
- **v7.1:** Pretty-Printing Enums
- **v7.2:** Event Probes (eprobes) pointer resolving
- **v7.3:** Explicit Typecasting, Nested Casting, \$current, & this_cpu_ptr()
- **v7.3:** Tracepoint BTF IDs exposed via tracefs
- **Upcoming:** wprobe structure offset resolution

BPF Tracing Support (v5.x)

BTF became foundational for enabling strongly-typed BPF hooks starting around v5.5.

- **v5.5** Raw Tracepoint Contexts

Accurate access to `raw_tp` contexts via BTF types.

- **v5.7 / v5.8** `PROG_TRACING`

Enable direct attachment to kernel functions with typed context, replacing legacy probe limitations.

- **v5.10** Structured Formatting Helpers

`bpf_seq_printf_btf` & `bpf_snprintf_btf` introduced to inspect and safely format structured data natively using BTF types.

Tracing Probes (Kprobe/Fprobe) Evolution (v6.x)

BTF fundamentally simplified kprobe and fprobe definitions, removing the need to memorize CPU register schemas.

⦿ v6.5 Function Parameter Access

Reference function parameters reliably by variable name rather than via architecture-specific registers.

⦿ v6.5 Typed `$retval`

Fetch the return value correctly styled using its BTF signature map.

⦿ v6.6 Struct Field Dereferencing

Access data structure fields easily via `ptr->field`.

⦿ v6.6 Module Support

BTF argument support securely extended to kernel module functions.

Usage Example: Function Parameters (v6.5)

Traditional Tracing (Without BTF)

Tracing `vfs_read()` parameters historically required referring directly to architecture-specific registers.

Modern Solution (With BTF)

Direct Variable & Meta-Variable Resolution

We can now use parameter variable names directly in our trace definitions, or leverage the powerful `$arg*` meta-variable to capture all function parameters instantly.

```
bash

# Fetch all arguments using $arg*
$ echo 'f vfs_read $arg*' >> dynamic_events

# The kernel automatically resolves BTF types
and names
$ cat dynamic_events

f:fprobes/vfs_read vfs_read file=file buf=buf
count=count pos=pos
```

Usage Example: Typed Returns (v6.5)

The `$retval` pseudo-variable automatically inherits the true return type of the traced function directly from its compiled **BTF (BPF Type Format)** metadata.

This architectural addition guarantees strong type safety during kernel return path analysis without requiring manual configuration or fragile type casting.

Automatic Void Verification

If a target function returns `void`, any system references to `$retval` are correctly and programmatically rejected.

Capture the typed return value of `vfs_read`:

```
  
# Register fprobe return tracer  
$ echo 'f vfs_read%return $retval' \  
>> dynamic_events  
  
# Verify auto-generated signature  
$ cat dynamic_events  
  
f:fprobes/vfs_read__return \  
vfs_read%return arg1=$retval
```

Usage Example: Struct Dereferencing (v6.6)

BTF ends the need for manual offset calculations to fetch deep struct fields.

It enables standard C-style pointer dereference syntax, safely traversing pointer structures within the Linux kernel.

Standardized syntax simplifies tracing setup and increases portability across kernel versions.

Dereference the 'prev' pointer to get the 'comm' char array:

```
$ echo 't sched_switch prev->comm:string' >>
dynamic_events
```

The same works alongside \$retval references:

```
$ echo 'f getname_flags%return
$retval->name:string' >> dynamic_events
```

Advanced Filtering, Eprobes, and Trace Events (v7.0+)

Applying BTF seamlessly into generic kernel tracing mechanisms.

- **v7.1** **Pretty-Printing Enums** — Trace output reliably resolves an enum parameter's integer value and displays its symbolic name using BTF parsing.
- **v7.2** **Event Probes (eprobes)** — Allows dereferencing eprobe pointers natively by employing names resolved through BTF.
- **v7.3** **Explicit BTF Typecasting** — Dynamically cast parameters to known structs.
- **v7.3** **Tracefs Exposure** — Tracepoint BTF IDs are exposed via tracefs allowing easy user-space tooling alignment.

Usage Example: Explicit BTF Typecast (v7.3)

You can directly cast an event field to an available BTF struct type and dereference its members inside dynamic event rules.

Format: (STRUCT[,ASGN])FIELD->MEMBER

Note:

An ,ASGN parameter can optionally be used to cast the field to a specific internal member's shared offset position rather than the start of the STRUCT.

Typecast arg1 to a struct file and fetch the f_mode field dynamically:

```
# Apply dynamic event rule with BTF typecast
$ echo 'p:myprobe my_func obj=(struct
file)arg1->f_mode' >> dynamic_events
```

Usage Example: Nested Type Casting (v7.3)

Because **BTF typecasts** are powerfully abstracted, they can evaluate dynamically nested rules to handle complex indirect pointers natively.

Instead of casting exactly from an event arg, you can perform typecasting deeply inside an already-resolved expression or memory offset variable.

Evaluation Format

(STRUCT[, ASGN]) (FETCHARG) -> MEMBER

Dynamic Event Definition

Deep nested BTF cast on task_struct at offset +8

```
# Register probe with deep nested typecast
$ echo 'p:myprobe my_func \
val=(struct task_struct)(+8($arg1))->pid' \
>> dynamic_events

# Evaluates task_struct address from (+8)
# and extracts final process ID field
```

Usage Example: Per-CPU Variable Fetching (v7.3)

Local CPU State Tracing

Accessing local per-CPU variables safely is often vital for troubleshooting complex runtime issues directly within the kernel.

TraceFS now explicitly supports these operations to read direct scalars and complex nested pointers without locking overhead.

Supported Core API

Native TrafeFS Operations

`this_cpu_read(VAR)`

Reads a direct scalar value safely from a per-CPU variable.

`this_cpu_ptr(VAR)`

Generates a pointer to dynamically dereference deeply typed struct fields.

Read a direct scalar value from a per-CPU variable:

```
$ echo 'p:tick tick_nohz_idle \
    val=this_cpu_read(tick_cpu_sched)' \
>> dynamic_events
```

Combine with nested BTF typecasting to dereference typed pointer fields:

```
$ echo 'p:tick tick_nohz_idle \
    rq=(struct rq)this_cpu_ptr(runqueues)\
->nr_running' \
>> dynamic_events
```

Usage Example: Typed \$current Task Mapping (v7.3)

The universal **\$current** pseudo-variable yields the address of the underlying `task_struct`.

Thanks to BTF, **\$current** implicitly inherits the type **struct task_struct ***.

No manual casting is needed when dereferencing nested members within tracing structures.

Context Specifics: Safely pull deeply nested task specifics cleanly out of context metrics without complex setup overhead.

```
# Trace task name and CPU flags without manual
casting

$ echo 'f vfs_read \
    task=$current->comm \
    cpu=$current->cpus_ptr->flags' \
    >> dynamic_events
```

Upcoming Features

Watchpoint Probes (wprobe)

Integration of BTF across more trace features continues.

The upcoming **watchpoint probes (wprobe)** feature introduces direct BTF structure offset resolution inside `set_wprobe` trigger configurations.

This allows developers to dynamically resolve structure offsets and start watching the memory of specific struct fields directly from the command line.

Setup a dummy watchpoint, trace `do_truncate`, and dynamically watch the `d_inode` field:

```
# Setup dummy watchpoint
$ echo 'w:watch_dentry rw@-1:8 value=+0($addr)' \
>> dynamic_events

# Trace do_truncate
$ echo 'f:truncate do_truncate dentry=$arg2' \
>> dynamic_events

# Watch d_inode field dynamically
$ echo 'set_wprobe:watch_dentry:(struct dentry)'\
'dentry->d_inode' >> \
events/fprobes/truncate/trigger
```

How BTF Simplifies Kernel Debugging

Democratized Tracing: BTF abstracts away architecture-specific registers and manual memory offset calculations.

Safer & More Standard Syntax: Enables familiar C-style pointer dereferencing (`ptr->field`) directly from tracefs.

Widespread Adoption: Expanding from core eBPF to kprobe, fprobe, eprobe, trace events, and now wprobe.

Improved Productivity: Significantly lowers the barrier to entry for debugging complex kernel data architectures.

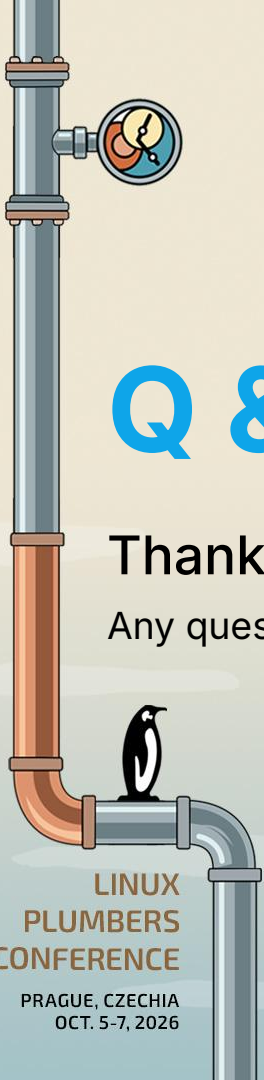
Future Ideas for BTF in Tracing

Advanced Structural Filtering: Using BTF for deep structural inspection inside trace filters
`filter="task->parent->pid == 123".`

Synthetic Events Expansion: Enforcing strong typing on synthetic event generation, histograms, and histograms variables.

Trace Event Injection: BTF-based structural type-checking and validation when injecting events back via `tracefs`.

Any ideas welcome!



Q & A

Thank you for listening!

Any questions?

LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA
OCT. 5-7, 2026