



Red Hat

Pluggable RV Monitors with BPF

Kernel Tech Talk - 17 September 2026

Gabriele Monaco

Senior Software Engineer

Kernel Real-Time

Kernel Tech Talk - 17 September 2026

Agenda

Pluggable RV Monitors with BPF

- ▶ Introduction
- ▶ Create a New Monitor
- ▶ Userspace API and Tools
- ▶ Up Next



Introduction



What is RV

Runtime Verification (RV) is a lightweight (yet rigorous) method that complements classical exhaustive verification techniques with a more practical approach for complex systems.¹

- ▶ Verify the Linux kernel works according to specification.
- ▶ Document kernel internals with graphical state machines.
- ▶ Enforce theoretical assumptions to pave the way for other tools or proofs.

The project was created and developed by Daniel Bristot de Oliveira.

¹ <https://docs.kernel.org/trace/rv/runtime-verification.html>



How Does it Work

- ▶ Monitor implements formal model.
- ▶ Hooks into the system events (tracepoints via kernel code).
- ▶ Compares flow against formal specification → React on violations.
- ▶ Does not *prove* the specification is correct (other tools can).

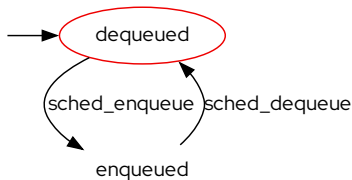


Figure: Example of a Deterministic Automaton (DA)



Use Cases

Currently upstream as in-kernel monitors

Scheduler Model

Theoretical foundation of RTLA (Real-Time Linux Analysis tool).

RTapp

Verify userspace is well configured for real-time use.

Deadline Models

Validate functional and timing behaviour of the DL scheduler.



Monitor Inclusion Workflow

- ▶ Come up with a model idea and code.
 - ▶ *A deterministic automaton for tasks..*
 - ▶ *Tasks alternate enqueue and dequeue → new tracepoints are correct.*
- ▶ Submit upstream for in-tree inclusion.
- ▶ What about niche models?
 - ▶ Domain-specific logic.



Monitor Inclusion Workflow

- ▶ Come up with a model idea and code.
 - ▶ *A deterministic automaton for tasks..*
 - ▶ *Tasks alternate enqueue and dequeue → new tracepoints are correct.*
- ▶ Submit upstream for in-tree inclusion.
- ▶ What about niche models?
 - ▶ Domain-specific logic.

Write pluggable monitors in BPF! ²

² [PATCH v2] rv: Add support for BPF monitors - <https://lore.kernel.org/lkml/20261001152042.124445-1-gmonaco@redhat.com>



RV Monitors in BPF

Test models without reboot.

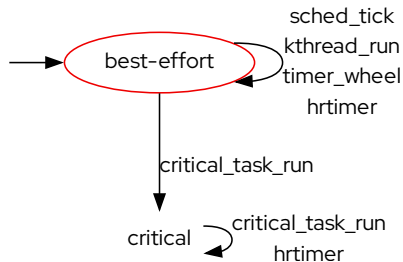
All BPF probe types freely available.

Same simple and lightweight model verification.



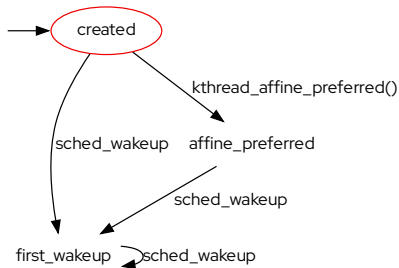
Example: Domain Specific Rule

- ▶ Mixed criticality system.
- ▶ CPUs start in *best-effort* state.
- ▶ CPUs where critical tasks run are *critical*.
- ▶ Kthreads, ticks, imprecise timers on BE.
- ▶ Critical CPUs cannot switch to BE.



Example: API Usage Validation

- ▶ Exported function with usage rule.
- ▶ *kthread_affine_preferred()* must be called before the first wakeup of the kthread.
- ▶ Validate all users (also out-of-tree).



Example: API Usage Validation

- ▶ Exported function with usage rule.
- ▶ *kthread_affine_preferred()* must be called before the first wakeup of the kthread.
- ▶ Validate all users (also out-of-tree).

```
digraph state_automaton {
  {node [shape = plaintext, style=invis, label=""]
    "__init_created"};
  {node [shape = ellipse] "created"};
  {node [shape = plaintext] "first_wakeup"};
  {node [shape = plaintext] "affine_preferred"};
  "__init_created" -> "created";
  "created" [label = "created"];
  "created" -> "affine_preferred" [
    label = "kthread_affine_preferred()"
  ];
  "created" -> "first_wakeup" [ label = "sched_wakeup" ];
  "first_wakeup" [label = "first_wakeup"];
  "first_wakeup" -> "first_wakeup" [ label = "sched_wakeup" ];
  "affine_preferred" [label = "affine_preferred"];
  "affine_preferred" -> "first_wakeup" [
    label = "sched_wakeup"
  ];
}
```



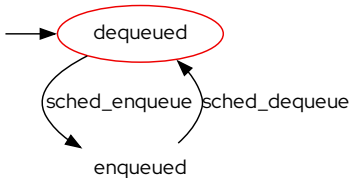
Create a New Monitor



Code Generation

- ▶ The `rvgen` tool converts the model representation to C source.
- ▶ Use `-b` to generate BPF code.
- ▶ Manual steps still needed to attach to kernel events.

```
python tools/verification/rvgen monitor -b -t per_task -c da -s tqueue.dot
```



Code Generation

tqueue.c (in-kernel)

```
#define RV_MON_TYPE RV_MON_PER_TASK
#include "tqueue.h"
#include <rv/da_monitor.h>

static void handle_sched_dequeue(void *data, /* XXX: fill header */)
{
    struct task_struct *p = /* XXX: how do I get p? */;
    da_handle_start_event(p, sched_dequeue_tqueue);
}
static void handle_sched_enqueue(void *data, /* XXX: fill header */)
{
    struct task_struct *p = /* XXX: how do I get p? */;
    da_handle_event(p, sched_enqueue_tqueue);
}

static int enable_tqueue(void)
{
    da_monitor_init();
    rv_attach_trace_probe("tqueue", /* XXX: tracepoint */, handle_sched_dequeue);
    rv_attach_trace_probe("tqueue", /* XXX: tracepoint */, handle_sched_enqueue);
    return 0;
}
static void disable_tqueue(void)
{
    rv_detach_trace_probe("tqueue", /* XXX: tracepoint */, handle_sched_dequeue);
    rv_detach_trace_probe("tqueue", /* XXX: tracepoint */, handle_sched_enqueue);
    da_monitor_destroy();
}
static struct rv_monitor rv_this = {
    .name = "tqueue", .enable = enable_tqueue, .disable = disable_tqueue,
};
```



Code Generation

tqueue.c (in-kernel)

```
#define RV_MON_TYPE RV_MON_PER_TASK
#include "tqueue.h"
#include <rv/da_monitor.h>

static void handle_sched_dequeue(void *data, /* XXX: fill header */)
{
    struct task_struct *p = /* XXX: how do I get p? */;
    da_handle_start_event(p, sched_dequeue_tqueue);
}
static void handle_sched_enqueue(void *data, /* XXX: fill header */)
{
    struct task_struct *p = /* XXX: how do I get p? */;
    da_handle_event(p, sched_enqueue_tqueue);
}

static int enable_tqueue(void)
{
    da_monitor_init();
    rv_attach_trace_probe("tqueue", /* XXX: tracepoint */, handle_sched_dequeue);
    rv_attach_trace_probe("tqueue", /* XXX: tracepoint */, handle_sched_enqueue);
    return 0;
}
static void disable_tqueue(void)
{
    rv_detach_trace_probe("tqueue", /* XXX: tracepoint */, handle_sched_dequeue);
    rv_detach_trace_probe("tqueue", /* XXX: tracepoint */, handle_sched_enqueue);
    da_monitor_destroy();
}
static struct rv_monitor rv_this = {
    .name = "tqueue", .enable = enable_tqueue, .disable = disable_tqueue,
};
```

Event handlers

Handlers registration



Code Generation

tqueue.c (BPF)

```
#define RV_MON_TYPE RV_MON_PER_TASK
#include "tqueue.h"
#include <rv/da_monitor.h>

SEC(/* XXX: tracepoint or other probe */)
int BPF_PROG(handle_sched_dequeue, /* XXX: fill header */)
{
    struct task_struct *p = /* XXX: how do I get p? */;
    da_handle_start_event(p, sched_dequeue_tqueue);
    return 0;
}

SEC(/* XXX: tracepoint or other probe */)
int BPF_PROG(handle_sched_enqueue, /* XXX: fill header */)
{
    struct task_struct *p = /* XXX: how do I get p? */;
    da_handle_event(p, sched_enqueue_tqueue);
    return 0;
}

static struct rv_monitor rv_this = {
    .enabled = 0,
};
```

Event handlers, registered by libbpf



Code Generation

tqueue.c (BPF)

```
#define RV_MON_TYPE RV_MON_PER_TASK
#include "tqueue.h"
#include <rv/da_monitor.h>

SEC("tp_btf/sched_dequeue_tp")
int BPF_PROG(handle_sched_dequeue, struct task_struct *tsk)
{
    if (!(BPF_CORE_READ(tsk, flags) & PF_EXITING))
        da_handle_start_event(tsk, sched_dequeue_tqueue);
    return 0;
}

SEC("tp_btf/sched_enqueue_tp")
int BPF_PROG(handle_sched_enqueue, struct task_struct *tsk)
{
    da_handle_event(tsk, sched_enqueue_tqueue);
    return 0;
}

static struct rv_monitor rv_this = {
    .enabled = 0,
};
```



Code Generation

tqueue.c (BPF)

```

#define RV_MON_TYPE RV_MON_PER_TASK
#include "tqueue.h"
#include <rv/da_monitor.h>

SEC("tp_btf/sched_dequeue_tp")
int BPF_PROG(handle_sched_dequeue, struct task_struct *tsk)
{
    if (!(BPF_CORE_READ(tsk, flags) & PF_EXITING))
        da_handle_start_event(tsk, sched_dequeue_tqueue);
    return 0;
}

SEC("tp_btf/sched_enqueue_tp")
int BPF_PROG(handle_sched_enqueue, struct task_struct *tsk)
{
    da_handle_event(tsk, sched_enqueue_tqueue);
    return 0;
}

static struct rv_monitor rv_this = {
    .enabled = 0,
};

```

```

enum states_tqueue {
    dequeued_tqueue, enqueued_tqueue,
    state_max_tqueue,
};

enum events_tqueue {
    sched_dequeue_tqueue, sched_enqueue_tqueue,
    event_max_tqueue,
};

struct automaton_tqueue {
    char state_names[state_max_tqueue][32];
    char event_names[event_max_tqueue][32];
    char function[state_max_tqueue][event_max_tqueue];
};

static const struct automaton_tqueue automaton_tqueue = {
    .state_names = { "dequeued", "enqueued" },
    .event_names = { "sched_dequeue", "sched_enqueue" },
    .function = {
        { INVALID_STATE, enqueued_tqueue },
        { dequeued_tqueue, INVALID_STATE },
    },
};

```



Automata Logic

- ▶ Most of the logic is in the header (e.g. `<rv/da_monitor.h>`).
- ▶ Reuse that in BPF as much as possible.

```
#if RV_MON_TYPE == RV_MON_PER_CPU
static DEFINE_PER_CPU(struct da_monitor, DA_MON_NAME);

static struct da_monitor *da_get_monitor(void)
{
    return this_cpu_ptr(&DA_MON_NAME);
}

#endif /* RV_MON_TYPE */

static void da_handle_event(enum events event)
{
    struct da_monitor da_mon = da_get_monitor();

    if (da_monitor_handling_event(da_mon))
        __da_handle_event_common(da_mon, event, id);
}
```



Automata Logic

- ▶ Most of the logic is in the header (e.g. <rv/da_monitor.h>).
- ▶ Reuse that in BPF as much as possible.

```
#ifdef __BPF__
#include "da_monitor_bpf.h"
#else /* __BPF__ */

#if RV_MON_TYPE == RV_MON_PER_CPU
static DEFINE_PER_CPU(struct da_monitor, DA_MON_NAME);

static struct da_monitor *da_get_monitor(void)
{
    return this_cpu_ptr(&DA_MON_NAME);
}

#endif /* RV_MON_TYPE */

#endif /* __BPF__ */
```



Automata Logic

- ▶ Most of the logic is in the header (e.g. `<rv/da_monitor.h>`).
- ▶ Reuse that in BPF as much as possible.

```
/* da_monitor_bpf.h */

#if RV_MON_TYPE == RV_MON_PER_CPU
struct {
    __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
    __uint(max_entries, 1);
    __type(key, __u32);
    __type(value, struct da_monitor);
} da_monitor_map SEC(".maps");

static struct da_monitor *da_get_monitor(void)
{
    __u32 key = 0;

    return bpf_map_lookup_elem(&da_monitor_map, &key);
}

#endif /* RV_MON_TYPE */
```



Userspace API and Tools



List Available Monitors

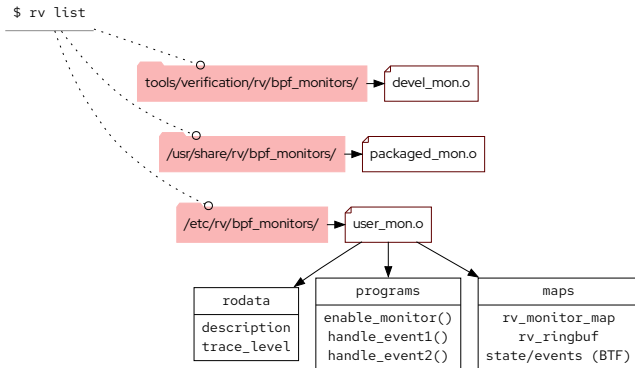
- ▶ `cat /sys/kernel/tracing/rv/available_monitors`
- ▶ `rv list`

<code>wwnr</code>	wakeup while not running per-task testing model. [OFF]
<code>sched</code>	container for several scheduler monitor specifications. [OFF]
- <code>sssw</code>	set state sleep and wakeup. [OFF]
- <code>nrp</code>	need resched preempts. [OFF]
- <code>snroc</code>	set non runnable on its own context. [OFF]
- <code>sco</code>	scheduling context operations. [OFF]
<code>rtapp</code>	Collection of monitors for detecting problems with real-time applications [OFF]
- <code>pagefault</code>	Monitor that RT tasks do not raise page faults [OFF]



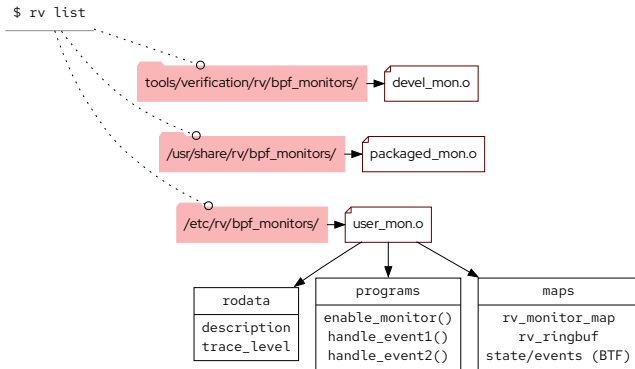
Distribute BPF Monitors

- ▶ BPF monitors are compiled objects in convenient locations.
- ▶ Well defined map names interface.



Distribute BPF Monitors

- ▶ BPF monitors are compiled objects in convenient locations.
- ▶ Well defined map names interface.



▶ `ls /etc/rv/bpf_monitors/`

`tqueue.o`

▶ `rv list`

```

wwnr      ... [OFF]
sched     ... [OFF]
- sssw    ... [OFF]
- nrp     ... [OFF]
- snroc    ... [OFF]
- sco     ... [OFF]
tqueue    enqueue/dequeue tasks. [OFF]
  
```



Start and Stop a Monitor

- ▶ `echo [0|1] > /sys/kernel/tracing/rv/monitors/MON/enable`
- ▶ `rv mon MON`



Start and Stop a Monitor

- ▶ ~~echo [0|1] > /sys/kernel/tracing/rv/monitors/MON/enable~~
- ▶ `rv mon MON`

Userspace handles registration: handlers + enable_monitor()



Show Monitor Events

- ▶ `echo 1 > /sys/kernel/tracing/events/rv/event_MON/enable`
- ▶ `echo 1 > /sys/kernel/tracing/events/rv/error_MON/enable`

```
ktimers/1-29 [001] 15.169563: event_wnnr: 29: running x switch_out -> not_running (final)
stress-ng-564 [001] 15.169606: event_wnnr: 29: not_running x switch_in -> running
ktimers/1-29 [001] 15.169626: error_wnnr: 29: event wakeup not expected in the state running
```



Show Monitor Events

- ▶ `echo 1 > /sys/kernel/tracing/events/rv/event_MON/enable`
- ▶ `echo 1 > /sys/kernel/tracing/events/rv/error_MON/enable`

```
ktimers/1-29 [001] 15.169563: event_wnnr: 29: running x switch_out -> not_running (final)
stress-ng-564 [001] 15.169606: event_wnnr: 29: not_running x switch_in -> running
ktimers/1-29 [001] 15.169626: error_wnnr: 29: event wakeup not expected in the state running
```

Cannot really have this in BPF



Show Monitor Events

- ▶ `echo 1 > /sys/kernel/tracing/events/rv/event_MON/enable` ▶ `rv mon -t MON`
- ▶ `echo 1 > /sys/kernel/tracing/events/rv/error_MON/enable`

<TASK>-PID	[CPU]	TYPE	ID	STATE	x	EVENT	-> NEXT_STATE	FINAL
<idle>-0	[012]	event	116	not_running	x	switch_in	-> running	N
108	[011]	error	52	running	x	wakeup		
<idle>-0	[003]	event	44	not_running	x	wakeup	-> not_running	Y



Show Monitor Events

- ▶ `echo 1 > /sys/kernel/tracing/events/rv/event_MON/enable` ▶ `rv mon -t MON`
- ▶ `echo 1 > /sys/kernel/tracing/events/rv/error_MON/enable`

<TASK>-PID	[CPU]	TYPE	ID	STATE	x	EVENT	-> NEXT_STATE	FINAL
<idle>-0	[012]	event	116	not_running	x	switch_in	-> running	N
108	[011]	error	52	running	x	wakeup		
<idle>-0	[003]	event	44	not_running	x	wakeup	-> not_running	Y

BPF can do this with a ringbuffer



Reactors

- ▶ Specify what to do on violations.
 - ▶ Currently used for reporting/analysis.
 - ▶ Available reactors (nop, printk, panic).
-
- ▶ `echo printk > /sys/kernel/tracing/rv/monitors/MON/reactors`
 - ▶ `rv mon MON -r printk`



Reactors

- ▶ Specify what to do on violations.
 - ▶ Currently used for reporting/analysis.
 - ▶ Available reactors (nop, printk, panic).
- ▶ `echo printk > /sys/kernel/tracing/rv/monitors/MON/reactors`
- ▶ `rv mon MON -r printk`

BPF cannot reuse reactors → can implement its own.



BPF Reactors

- ▶ `rv mon MON -r printk`
- ▶ Link `bpf_rv_react()` in-memory before load.
- ▶ Works with any program type.
- ▶ Same discovery logic as monitors.

```
/* da_monitors_bpf.h */
__noinline __weak void bpf_rv_react(char *msg) { }

#define rv_react(mon, fmt, ...) \
({ \
    char ____msg[256]; \
    if (BPF_SNPRINTF(____msg, sizeof(____msg), \
                     fmt, ##__VA_ARGS__) > 0) \
        bpf_rv_react(____msg); \
})

/* bpf_reactors/printk.c */

void bpf_rv_react(char *msg)
{
    bpf_printk("%s", msg);
}

static char DESCRIPTION[] SEC(".rodata.description") =
    "prints the exception msg to the trace buffer.;"
```



Performance Comparison

In-kernel vs BPF monitors

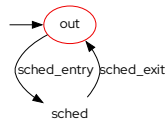
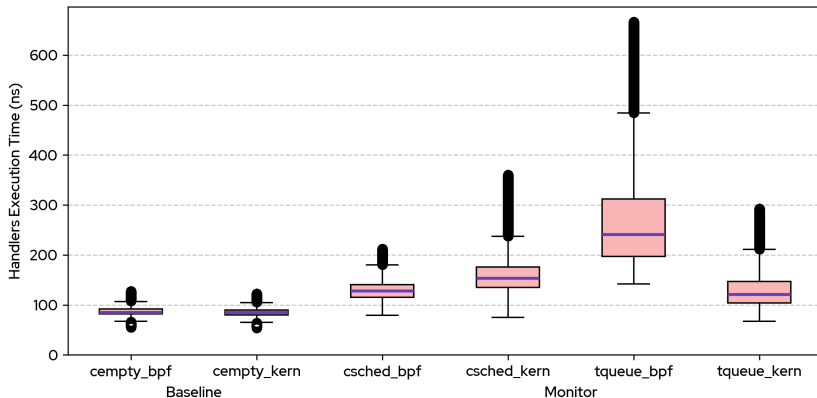


Figure: csched (per-cpu)

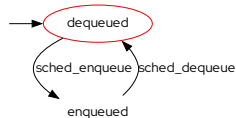


Figure: tqueue (per-task)

Figure: Lower is better. VM running RV and stress-ng on an Intel(R) i7-1270P.



Up Next



Up Next

- ▶ The `rv` userspace tool only supports DA.
- ▶ Support other monitor classes besides DA in BPF.
 - ▶ Linear Temporal Logic.
 - ▶ Hybrid/Timed Automata.



Thank you

Any questions?

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.



[linkedin.com/company/red-hat](https://www.linkedin.com/company/red-hat)



[facebook.com/redhatinc](https://www.facebook.com/redhatinc)



[youtube.com/user/RedHatVideos](https://www.youtube.com/user/RedHatVideos)



twitter.com/RedHat



Appendix

Backup slides

RV Benchmarking Code

```
#ifndef __BPF__
#include <linux/ftrace.h>
#include <linux/trace_clock.h>

struct rv_bench_ctx {
    const char *mod;
    u64 start;
};
#else

#define trace_printk          bpf_printk
#define ktime_get_mono_fast_ns bpf_ktime_get_ns
#endif

static inline void __rv_bench_end(struct rv_bench_ctx *ctx)
{
    trace_printk("%s: %llu ns\n", ctx->mod,
        ktime_get_mono_fast_ns() - ctx->start);
}

#define RV_BENCH() \
    struct rv_bench_ctx __rv_start __attribute__(( \
        cleanup(__rv_bench_end))) = { .mod = __stringify(MONITOR_NAME), \
        .start = ktime_get_mono_fast_ns() }
```

```
int BPF_PROG(handle_sched_enqueue,
             struct task_struct *tsk)
{
    RV_BENCH();
    da_handle_event(tsk, sched_enqueue);
    return 0;
}

...

/* in-kernel */
static void handle_sched_enqueue(void *data,
                                struct task_struct *tsk,
                                int cpu)
{
    RV_BENCH();
    da_handle_event(tsk, sched_enqueue);
}
```

