BPF Shadow Stack

Current Kernel Stacks

Thread stack size:
  X86_64: 16KB
  arm64: 64KB
  S390: 4 * PAGE_SIZE

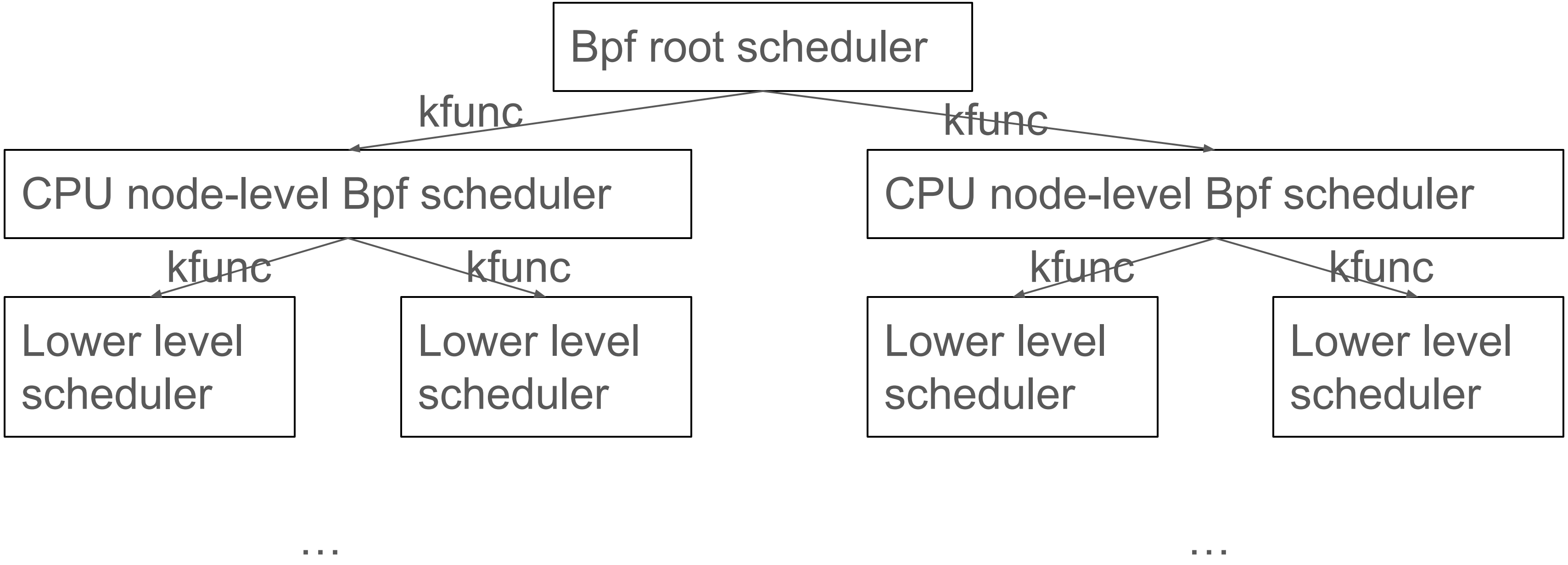
X86_64 IRQ stack size: 16KB

X86_64 Exception (NMI) stack size: 8KB

BPF programs: 512B + some additional stack spaces in jit and intepreter.

BPF Shadow Stack Use Case

Use case: Tracing, Sched-ext at different cgroup levels, bpf libraries, etc.



Approach 1: Additional Parameter to BPF Prog

```
__bpf_prog_run():
  frame_ptr = bpf_allocate_stack(prog)
  // frame_ptr will be the third argument for bpf
prog entry point.
  bpf_dispatcher_fn(ctx, insni, frame_ptr,
bpf_func);
  bpf_free_stack(prog, frame_ptr)

do_jit():
  if (arena_vm_start)
      push_r12(&prog);
  push_callee_regs(&prog, callee_regs_used);
+ if (!tail_call_reachable) {
+    emit_mov_reg(&prog, true, X86_REG_R9,
BPF_REG_3);
+    push_r9(&prog);
+ }
```

```
arch_prepare_bpf_trampoline():
 invoke_bpf_prog():
  +      /* call bpf_shadow_stack_alloc */
  +      /* arg1: mov rdi, prog */
  +      emit_mov_imm64(&prog, BPF_REG_1, (long) p >> 32, (u32) (long) p);
  +      if (emit_rsb_call(&prog, bpf_shadow_stack_alloc, image + (prog - (u8 *)rw_image)))
  +              return -EINVAL;
  +      emit_stx(&prog, BPF_DW, BPF_REG_FP, BPF_REG_0, -shadow_stack_off);
         …
  +      /* arg3: shadow_stack for jit */
  +      if (p->jited)
  +              emit_mov_reg(&prog, true, BPF_REG_3, BPF_REG_0);
        /* call JITed bpf program or interpreter */
        if (emit_rsb_call(&prog, p->bpf_func, image + (prog - (u8 *)rw_image)))
            return -EINVAL;
        ...
  +      /* call bpf_shadow_stack_free */
  +      emit_mov_imm64(&prog, BPF_REG_1, (long) p >> 32, (u32) (long) p);
  +      emit_ldx(&prog, BPF_DW, BPF_REG_2, BPF_REG_FP, -shadow_stack_off);
  +      if (emit_rsb_call(&prog, bpf_shadow_stack_free, image + (prog - (u8 *)rw_image)))
```

Additional Parameter to Prog

- Tailcall and extension programs do not really work as those programs are not triggered directly through trampoline invoke_bpf_prog or __bpf_prog_run.

- For tailcall and extension programs, more complicated implementation is possible when tailcall and extension programs enabled in jit. For example, do alloc and free in jit when those programs are enabled.

Approach 2: Implement In JIT

```
do_jit(struct bpf_prog *bpf_prog …)
+  stack_depth = 0; // hack! enable shadow stacking
  …
                if (arena_vm_start)
                        push_r12(&prog);
              push_callee_regs(&prog, callee_regs_used);
+               /* save r9 */
+               push_r9(&prog);
       }
       if (arena_vm_start)
              emit_mov_imm64(&prog, X86_REG_R12,
                           arena_vm_start >> 32, (u32) arena_vm_start);

+      err = emit_shadow_stack_alloc(&prog, bpf_prog, image,
temp);
+      if (err)
+             return err;
+
…
```

```
       for (i = 1; i <= insn_cnt; i++, insn++) {
+             if (src_reg == BPF_REG_FP) {
+                    pop_r9(&prog); push_r9(&prog); src_reg = X86_REG_R9;
+             }
+
+             if (dst_reg == BPF_REG_FP) {
+                    pop_r9(&prog); push_r9(&prog); dst_reg = X86_REG_R9;
+             }
+
              switch (insn->code) {
                     /* ALU */
@@ -2319,10 +2480,17 @@ st:                   if (is_imm8(insn->off))
                     seen_exit = true;
                     /* Update cleanup_addr */
                     ctx->cleanup_addr = proglen;
+
+                    err = emit_shadow_stack_free(&prog, bpf_prog, image +
addrs[i - 1], temp);
+                    if (err) return err;
+                    pop_r9(&prog);
                     if (bpf_prog->aux->exception_boundary) {
```

Emit Shadow Stack Alloc/Free in JIT

```
+static int emit_shadow_stack_alloc(...)
+{
+       /* save R1-R5 parameters to preserve original bpf args. */
+       emit_mov_reg(pprog, true, X86_REG_R9, BPF_REG_1);
+       push_r9(pprog);
+       ...
+       emit_mov_imm64(pprog, BPF_REG_1, (long) bpf_prog >>
32, (u32) (long) bpf_prog);
+       if (bpf_prog->sleepable)
+               func = (u8 *)bpf_shadow_stack_alloc;
+       else
+               func = (u8 *)bpf_shadow_stack_alloc_sleepable;
+       emit_call(pprog, func, …);
+       /* restore R1-R5 */
+       ...
+       /* Save the frame pointer to the stack  */
+       emit_mov_reg(pprog, true, X86_REG_R9, BPF_REG_0);
+       push_r9(pprog);
+
+       return 0;
+}
```

```
+/* call bpf_shadow_stack_free function. Preserve r0-r5 registers. */
+static int emit_shadow_stack_free(...)
+{
+       pop_r9(pprog);
+       push_r9(pprog);
+       /* X86_REG_R9 holds the shadow frame pointer */
+       emit_mov_reg(pprog, true, AUX_REG, X86_REG_R9);
+       /* save reg 0-5 to preserve original values */
+       emit_mov_reg(pprog, true, X86_REG_R9, BPF_REG_0);
+       push_r9(pprog);
+       ...
+       emit_mov_imm64(pprog, BPF_REG_1, (long) bpf_prog >> 32, (u32)
(long) bpf_prog);
+       emit_mov_reg(pprog, true, BPF_REG_2, AUX_REG);
+       if (bpf_prog->sleepable)
+               func = (u8 *)bpf_shadow_stack_free;
+       else
+               func = (u8 *)bpf_shadow_stack_free_sleepable;
+       emit_call(pprog, func, ...);
+       /* restore reg 0-5 to preserve original values */
+       ...
+       return 0;
+}
```

Shadow Stack Alloc/Free for Non-Sleepable

```
+static DEFINE_PER_CPU_PAGE_ALIGNED(u8, bpf_shadow_stack[8192]);
+static DEFINE_PER_CPU(atomic_t, bpf_shadow_frame_off);
+void * notrace bpf_shadow_stack_alloc(struct bpf_prog *prog)
+{
+       int stack_depth = round_up(prog->aux->stack_depth, 16);
+       atomic_t *frame_off_ptr;
+       u8 *stack_base;
+       int off;
+       if (!stack_depth) return NULL;
+       frame_off_ptr = this_cpu_ptr(&bpf_shadow_frame_off);
+       stack_base = this_cpu_ptr(&bpf_shadow_stack[0]);
+       off = atomic_add_return(stack_depth, frame_off_ptr);
+       return stack_base + 8192 - off + stack_depth;
+}
+void notrace bpf_shadow_stack_free(struct bpf_prog *prog, void *shadow_frame)
+{
+       int stack_depth = round_up(prog->aux->stack_depth, 16);
+       atomic_t *frame_off_ptr;
+       if (!stack_depth) return;
+       frame_off_ptr = this_cpu_ptr(&bpf_shadow_frame_off);
+       atomic_sub(stack_depth, frame_off_ptr);
+}
```

Prog A Start

Prog B start

Prog C start

Prog C end

Prog B end

Prog A end

Shadow Stack Alloc/Free for Sleepable

```
+void * notrace bpf_shadow_stack_alloc_sleepable(struct bpf_prog *prog)
+{
+	int stack_depth = prog->aux->stack_depth;
+	void *shadow_stack;
+
+	if (!stack_depth) return NULL;
+	shadow_stack = kmalloc(round_up(stack_depth, 16), __GFP_NORETRY);
+	if (!shadow_stack) return NULL;
+	return shadow_stack + round_up(stack_depth, 16);
+}
+
+void notrace bpf_shadow_stack_free_sleepable(struct bpf_prog *prog, void *shadow_frame)
+{
+	int stack_depth = prog->aux->stack_depth;
+	void *shadow_stack;
+
+	if (!shadow_frame) return;
+	shadow_stack = shadow_frame - round_up(stack_depth, 16);
+	kfree(shadow_stack);
+}
```

Prog A Start
Prog B start
Prog C start
Prog A end
Prog B end
Prog C end

Approach 3: Allocate Per-cpu/Per-program Stack

- For each program, allocate per-cpu show stack with stack size.
- Note that on the same cpu, the same bpf program cannot have recursion and this is guaranteed by bpf infrastructure.
- The per-cpu stack allocation allows progs running on different cpus concurrently.
- Note that we may have quite some waste on memory if the number of cpus are large and only one or smaller number of instances of the program is running at any time.

Other issues and next steps

- Performance evaluation
- Approach 2: Per-cpu stack size. Allocate 4 pages? Dynamically allocate more stack size if necessary and free the old one once it is not used any more?
- Approach 2: Per-cpu stack potential overflow. Guard page?
- Approach 2: Optimization for non-sleepable programs (e.g., inlining alloc/free)
- Approach 3: Potential more memory is needed, esp. for large number of cpus in the system.
- alloca by bpf program: allocate from preallocated stack space or kmalloc, stack pointer vs. frame pointer.
- Add a flag for BPF_PROG_LOAD to enable the shadow stack for bpf program?