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

VIRGINIA TECH

Overflowing the Kernel Stack with BPF

Sai Roop Somaraju <sairoop@vt.edu>
Siddharth Chintamaneni <sidchintamaneni@vt.edu>
Dan Williams <djwillia@vt.edu>

BPF and kernel safety

- BPF programs enable applications to extend the kernel's functionalities at runtime, all while ensuring stability and security.

- Guaranteed safety is made possible by the verifier engine which statically verifies BPF code.

BPF verifier and BPF runtime

- However, The verifier inherently relies on certain assumptions regarding the runtime execution environment, and these assumptions are essential to maintain safety.

- One such assumption is the availability of stack space to run the BPF program.

Agenda

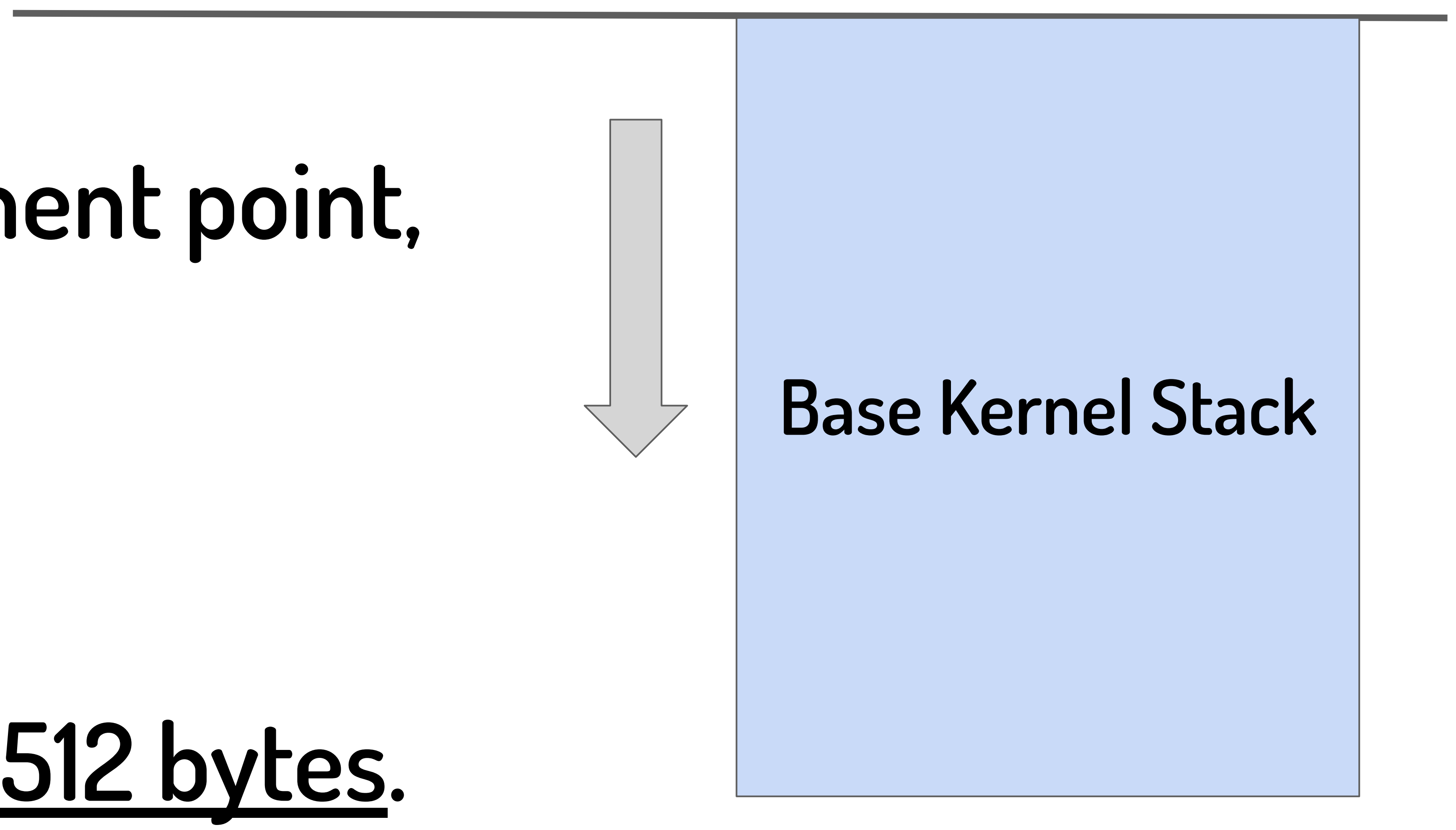
- BPF program attachment and its interaction with the stack
- Stack overflow due to BPF program attachment
- Stack overflow due to uncontrolled BPF program nesting
- Discussion on Probable Solutions and Related Questions
- Summary

BPF program interaction with the Kernel Stack

- When a verified BPF program executes from an attachment point, it typically inherits the existing kernel process stack.

- The stack space available to a BPF program is limited to 512 bytes.

- When a helper function or a kfunc is called from within the BPF program, it extends the same stack further into the kernel.

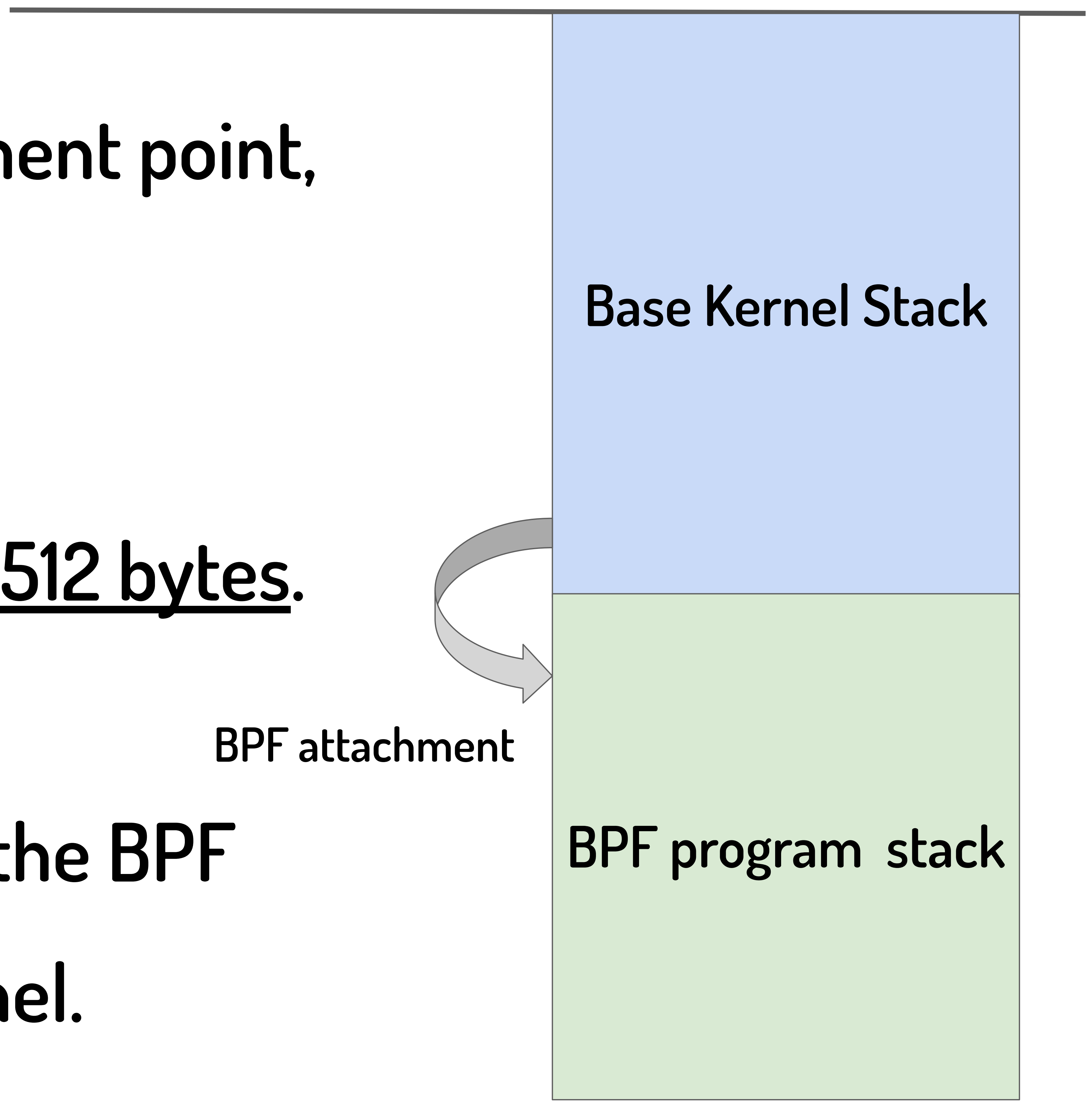


BPF program interaction with the Kernel Stack

- When a verified BPF program executes from an attachment point, it typically inherits the existing kernel process stack.

- The stack space available to a BPF program is limited to <u>512 bytes</u>.

- When a helper function or a kfunc is called from within the BPF program, it extends the same stack further into the kernel.

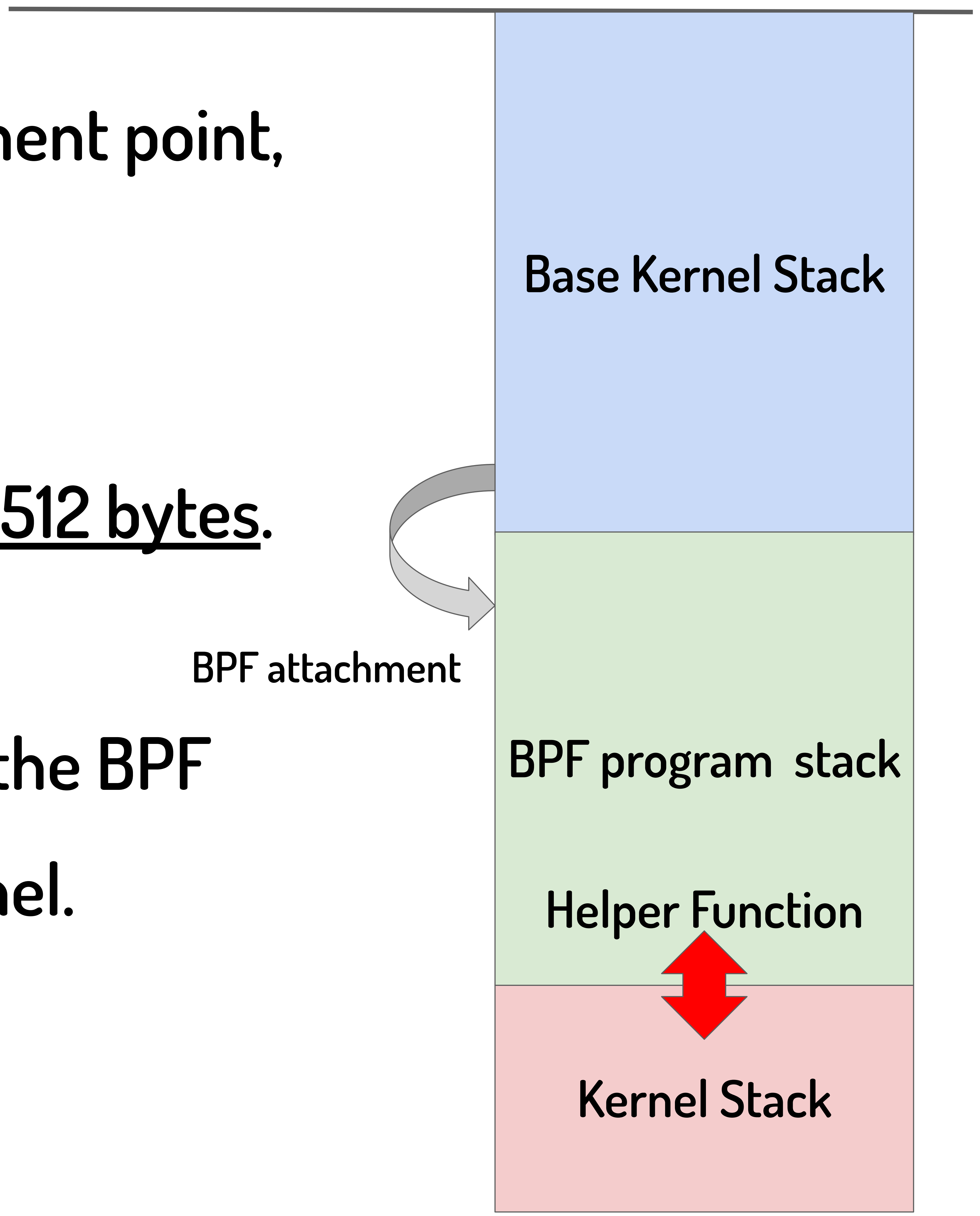


BPF program interaction with the Kernel Stack

- When a verified BPF program executes from an attachment point, it typically inherits the existing kernel process stack.

- The stack space available to a BPF program is limited to <u>512 bytes</u>.

- When a helper function or a kfunc is called from within the BPF program, it extends the same stack further into the kernel.

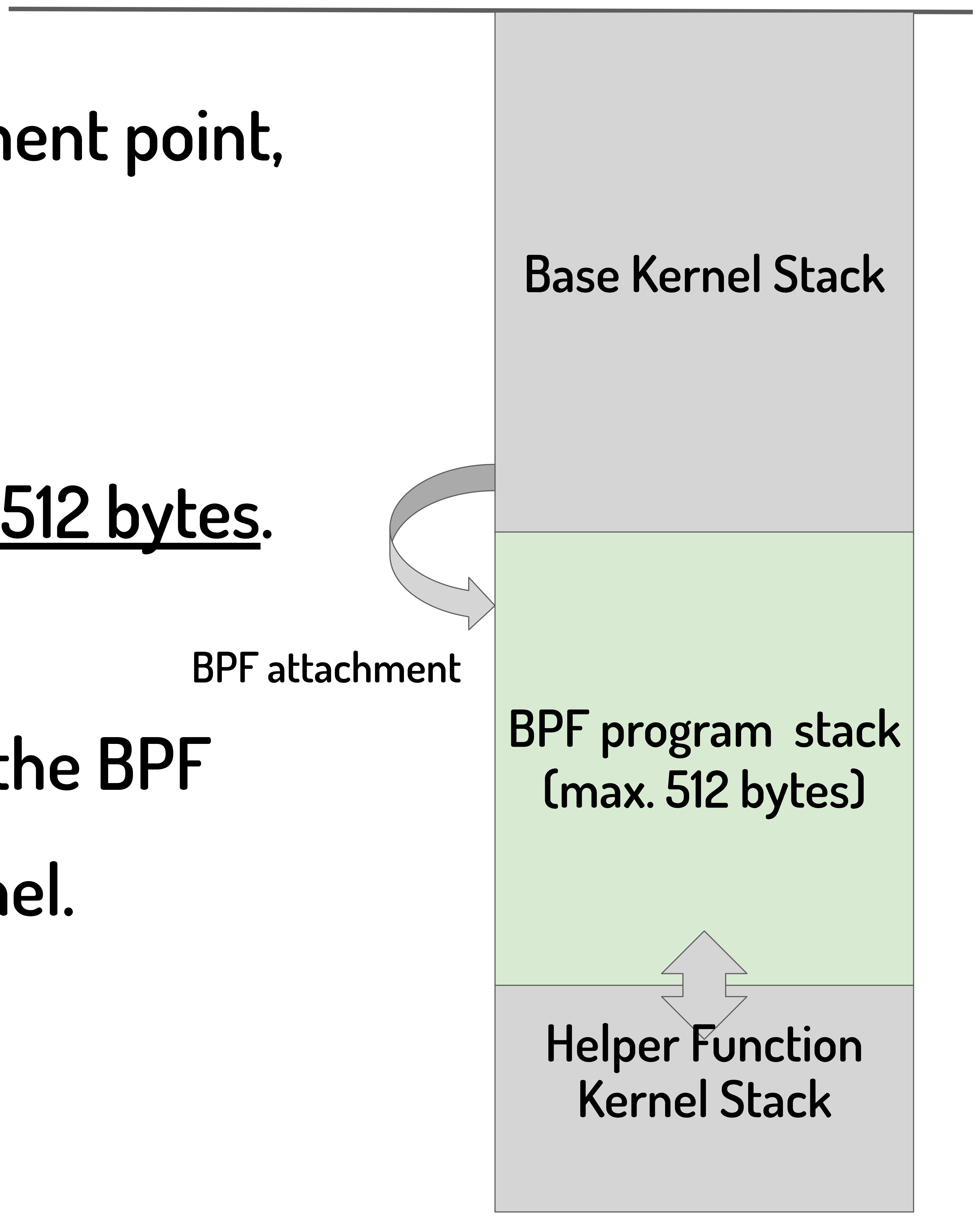


BPF program interaction with the Kernel Stack

- When a verified BPF program executes from an attachment point, it typically inherits the existing kernel process stack.

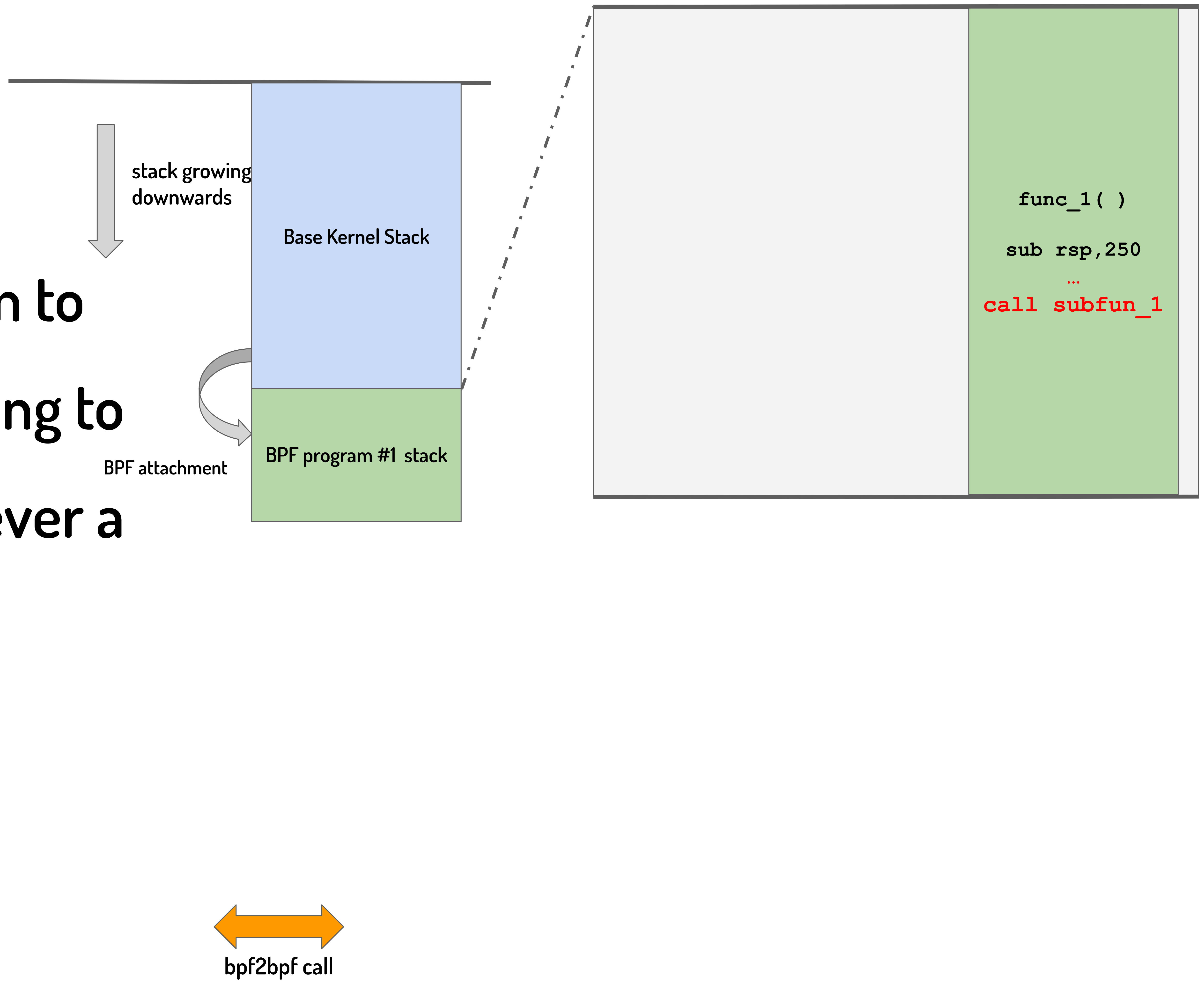
- The stack space available to a BPF program is limited to <u>512 bytes</u>.

- When a helper function or a kfunc is called from within the BPF program, it extends the same stack further into the kernel.

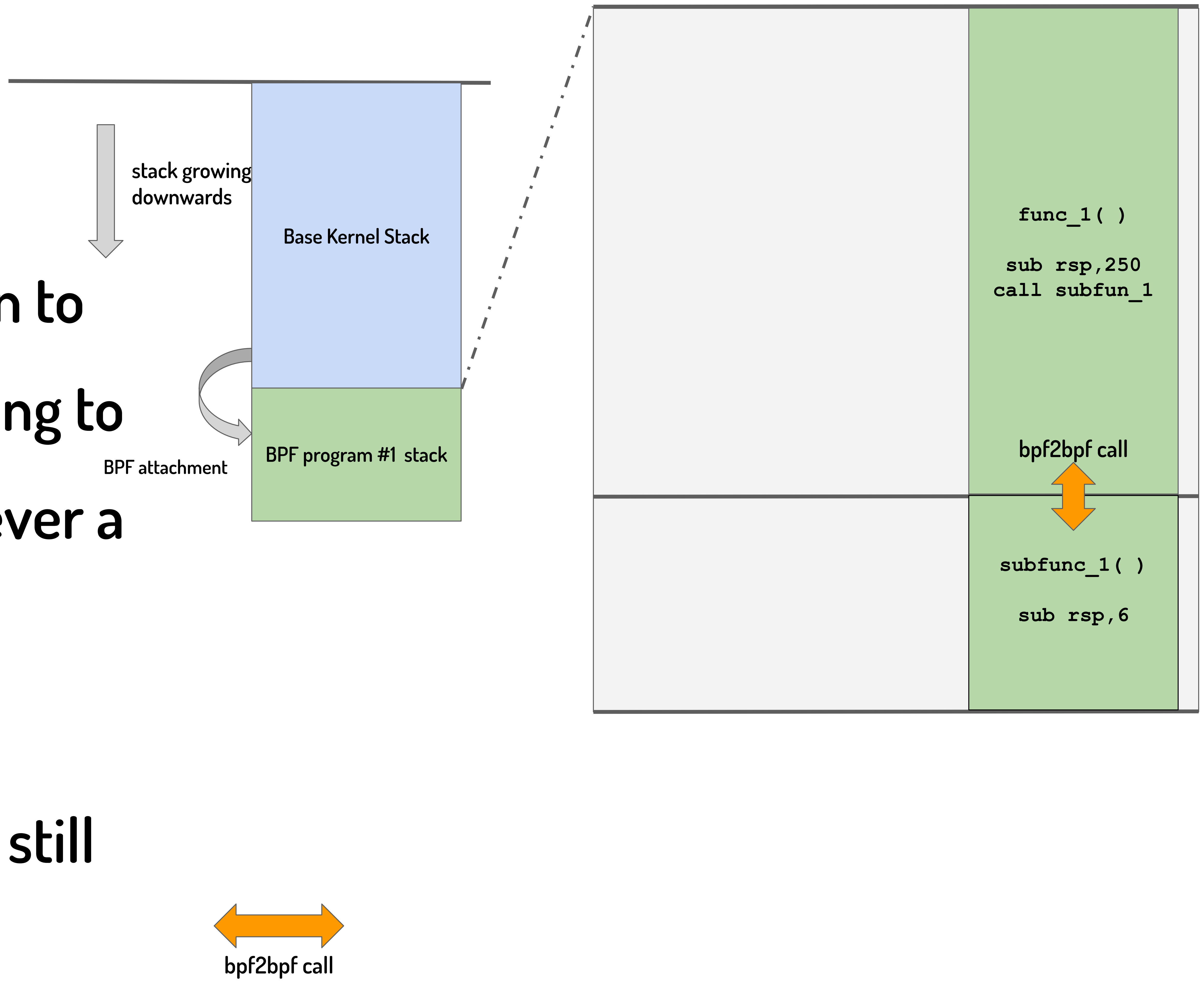


BPF-To-BPF calls

- BPF-to-BPF calls introduce a feature akin to function calls within BPF programs, leading to the creation of a new stack frame whenever a function call is initiated.



BPF-To-BPF calls

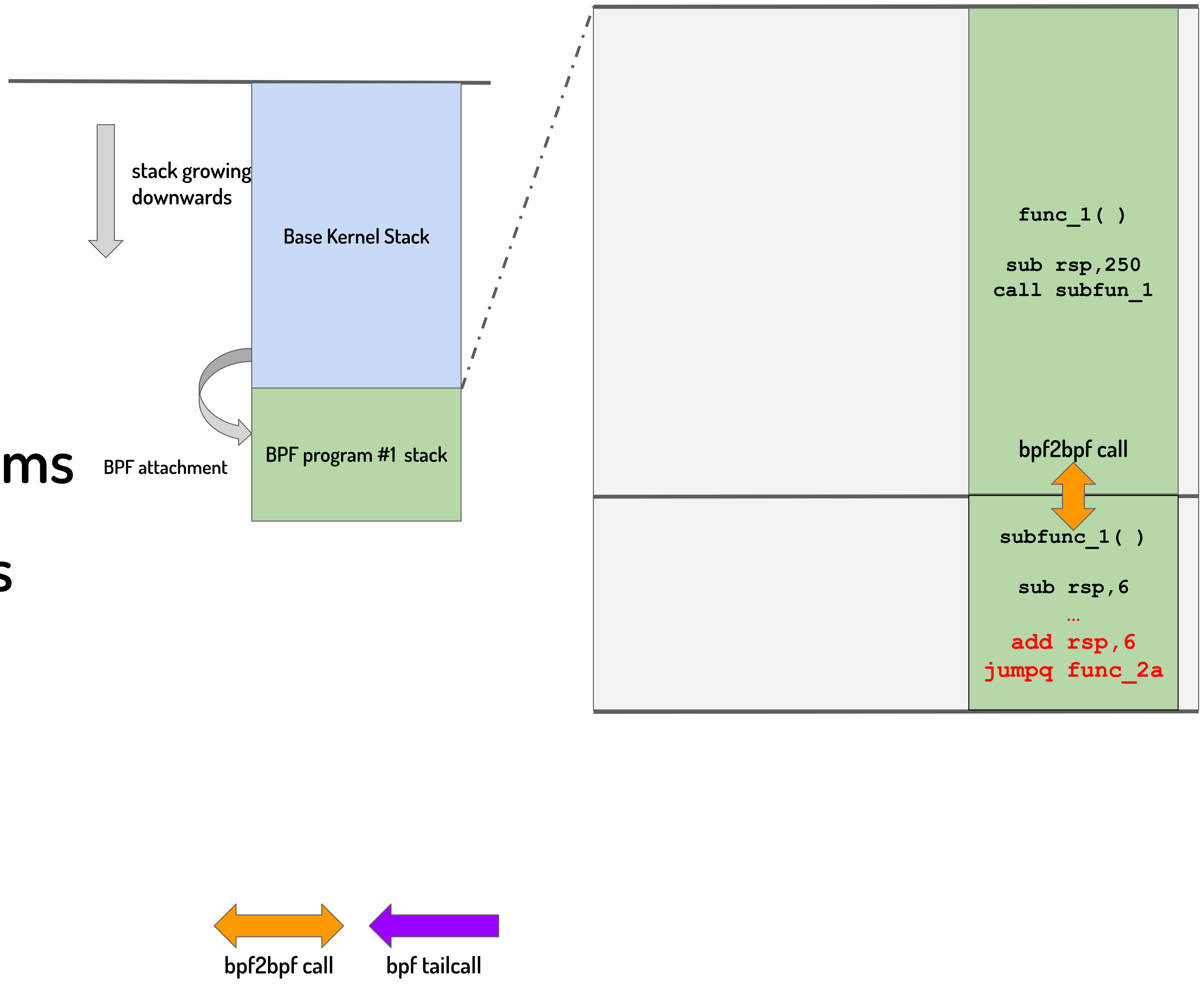


- BPF-to-BPF calls introduce a feature akin to function calls within BPF programs, leading to the creation of a new stack frame whenever a function call is initiated.

- The stack space for each BPF program is still limited to a maximum of 512 bytes.

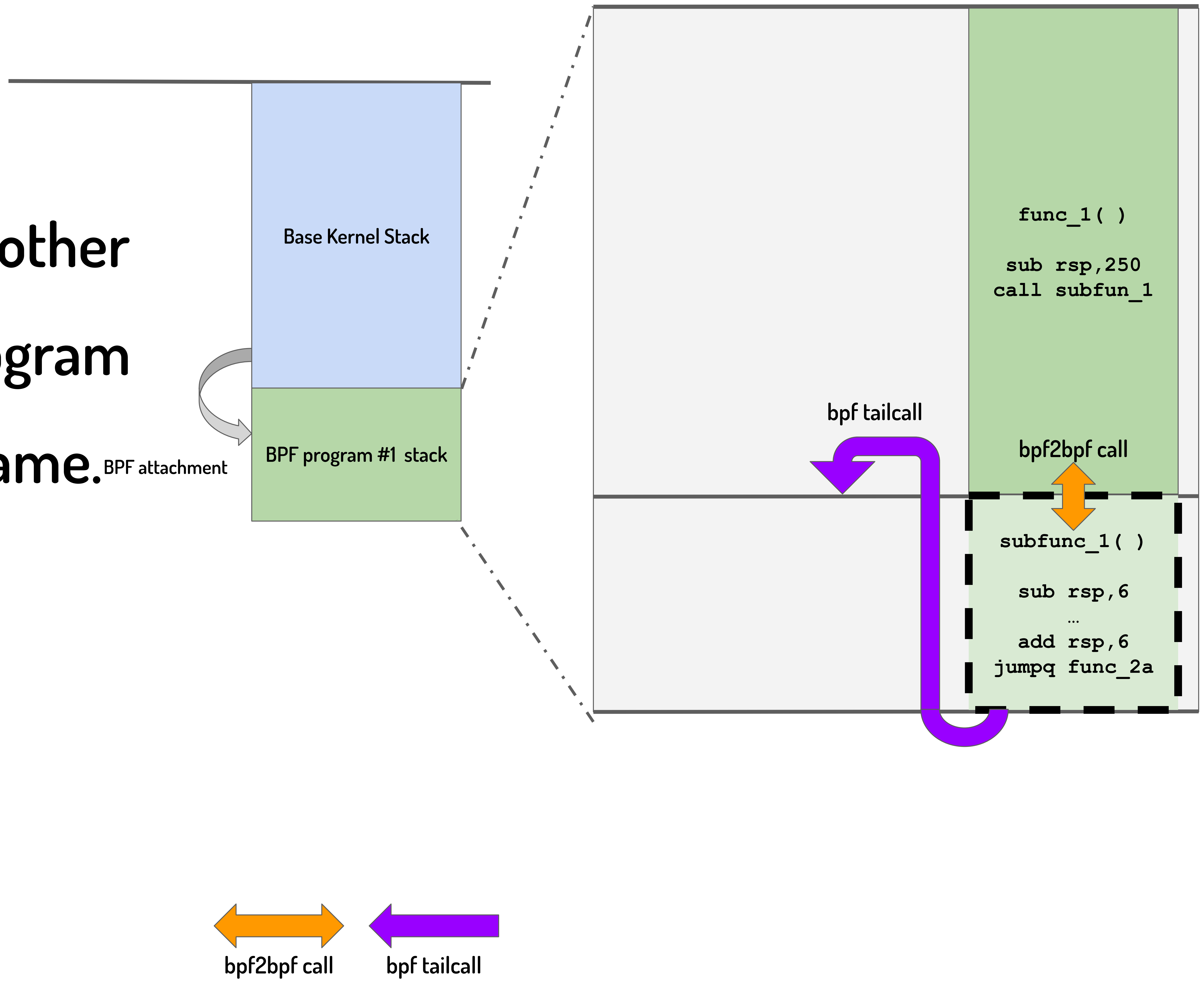
BPF tailcalls

- Tail calls allows a BPF program to call an another BPF program and the caller BPF program will <u>reuse</u> the callee BPF programs stack frame. Each BPF tail call program is verified individually.



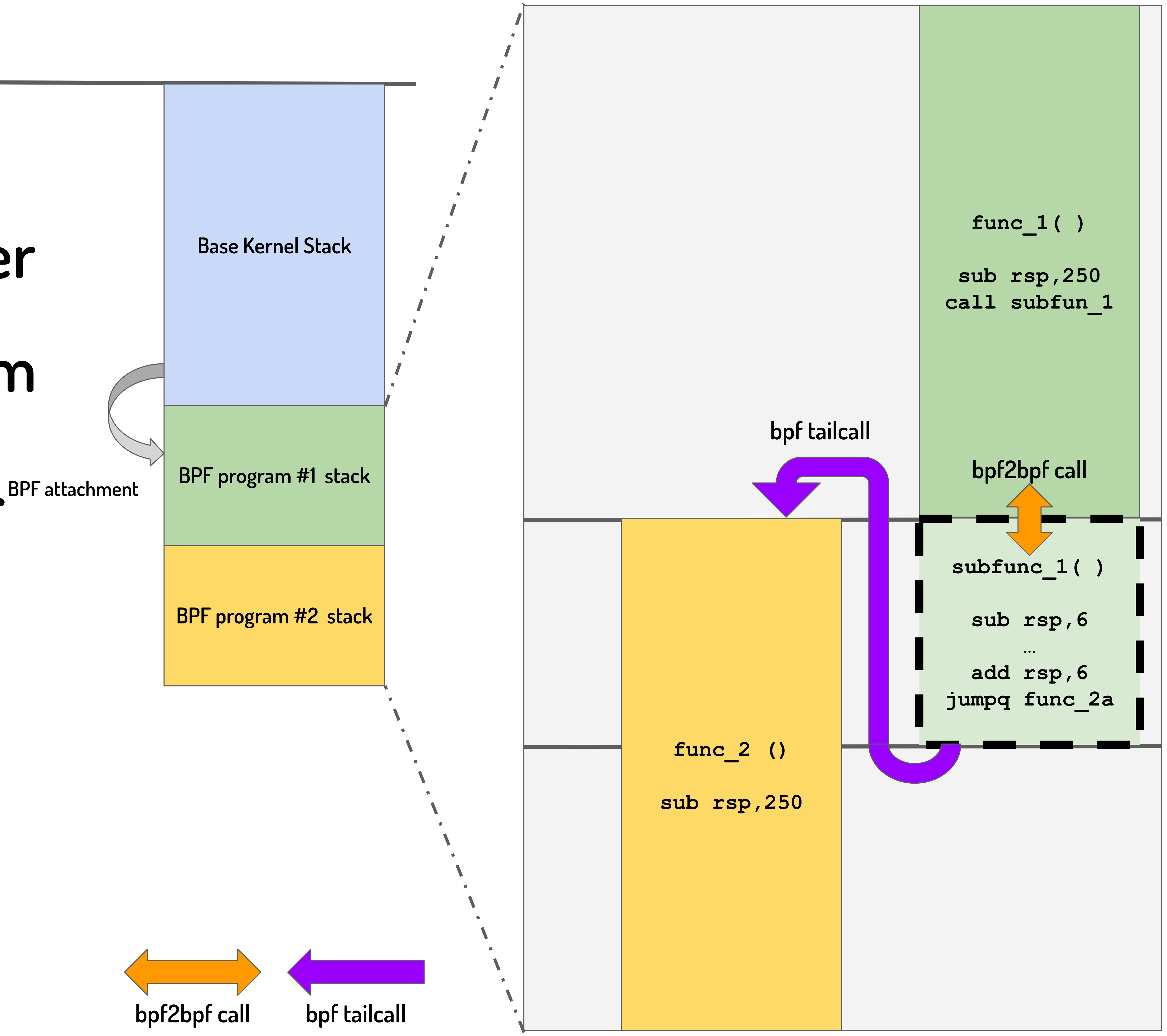
BPF tailcalls

- Tail calls enable a BPF program to call another BPF program, wherein the caller BPF program reuses the callee BPF program's stack frame. Each BPF tail call program is verified individually.



BPF tailcalls

- Tail calls enable a BPF program to call another BPF program, wherein the caller BPF program reuses the callee BPF program's stack frame. Each BPF tail call program is verified individually.

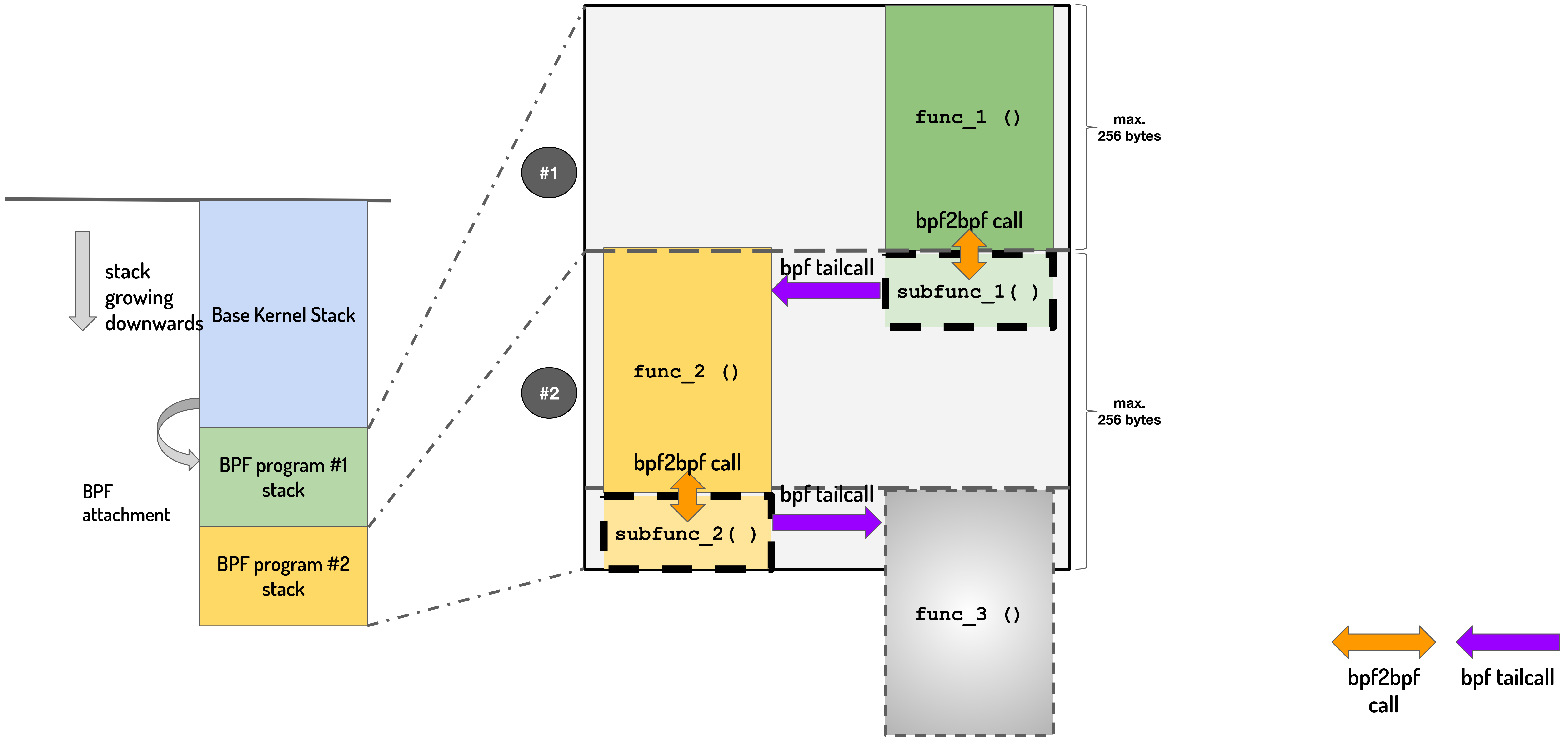


BPF tail calls and BPF-to-BPF calls: BPF checks and limits

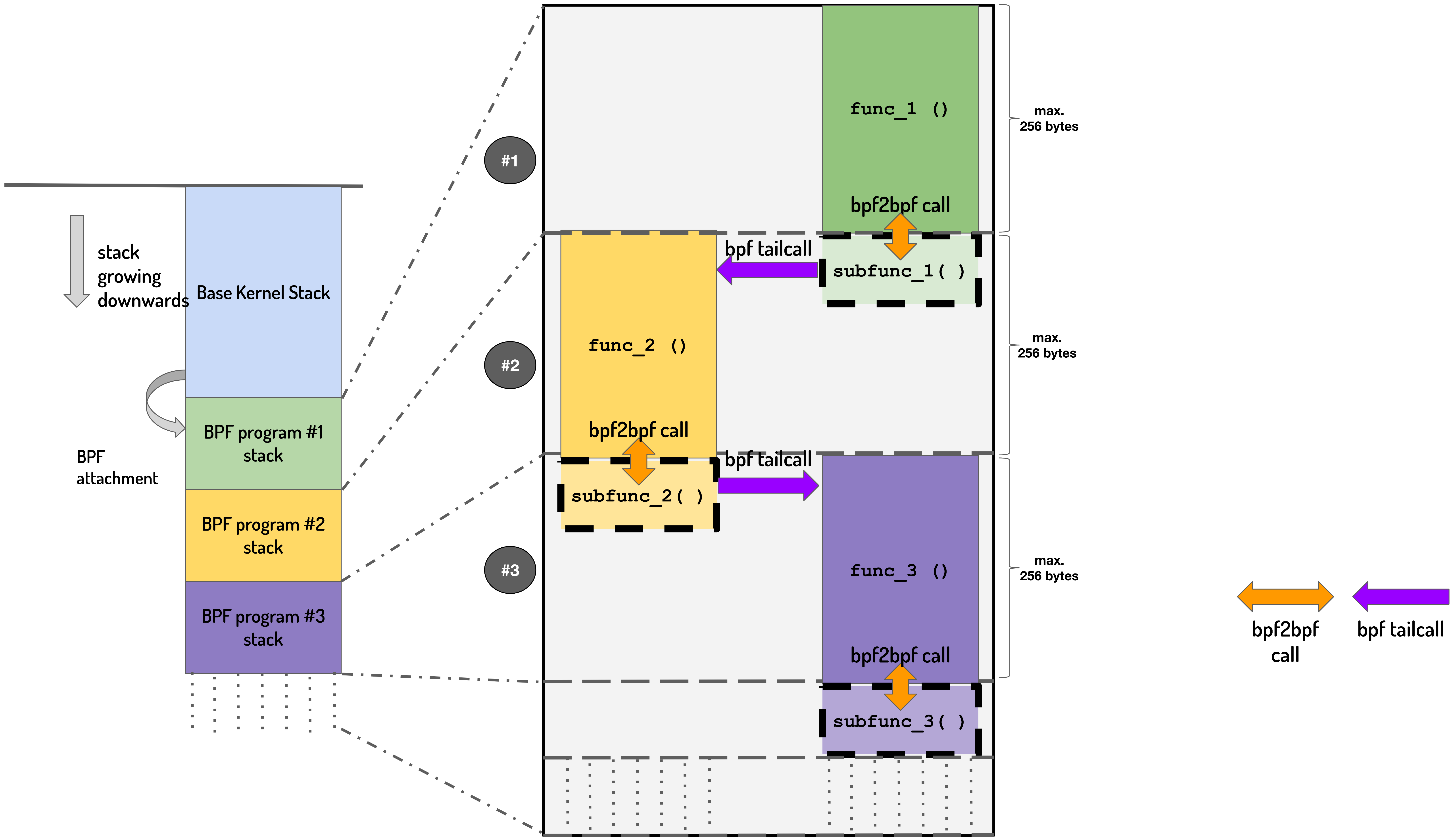
- If a tail call program is invoked from a BPF-to-BPF call function, the verifier restricts each BPF program's stack size to 256 bytes, as opposed to the standard 512 bytes.

- At runtime, the Just-In-Time (JIT) compiler limits the number of tail calls to no more than 33 tailcalls.

Writing a sizeable BPF program of size 8 KB



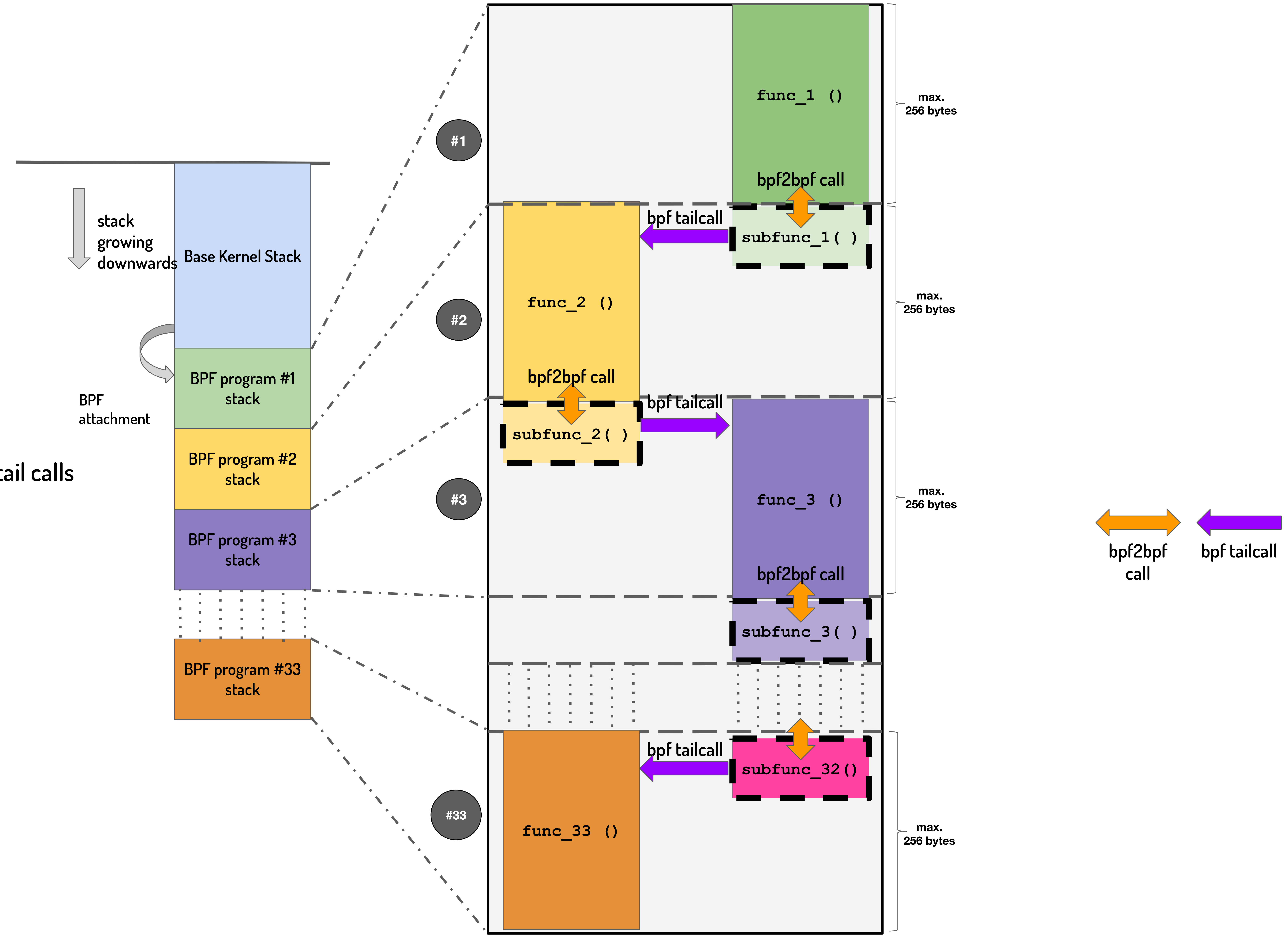
Writing a sizeable BPF program of size 8 KB



Writing a sizeable BPF program of size 8 KB

Each BPF program of *almost* ~256 bytes * #33 tail calls

= **~8,448 bytes**



BPF verifier assumptions about kernel's stack runtime

Therefore, here the verifier relies on two critical assumptions about the kernel's runtime to restrict the depth of verified BPF programs:

1. Kernel stack will always have 8 KB of stack space available for a BPF program to run.
2. The total size of the BPF program's kernel stack and the stack for any helper functions it calls will be less than 8 KB.

Agenda

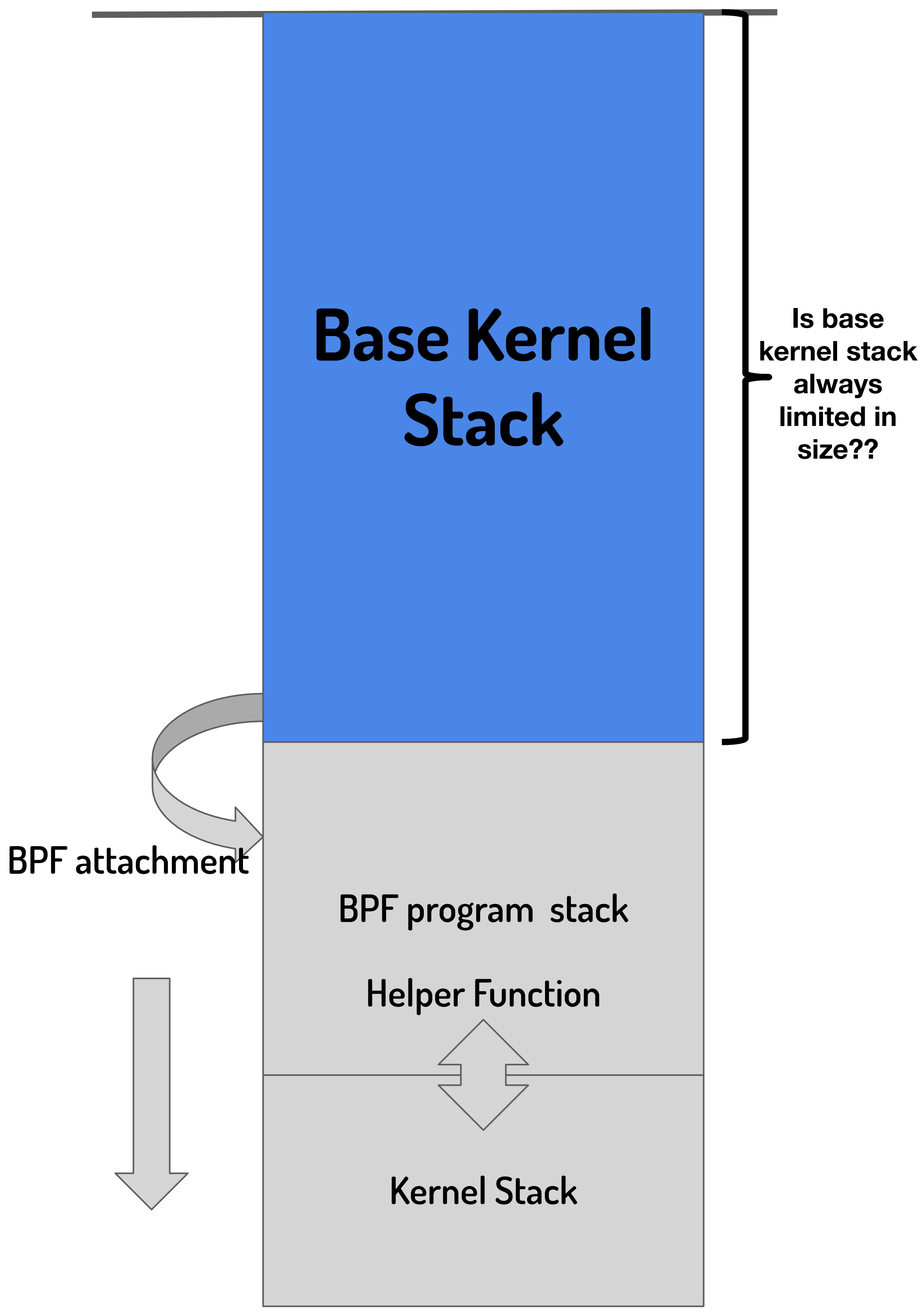
- BPF program attachment and its interaction with the stack
- Stack overflow due to BPF program attachment
- Stack overflow due to uncontrolled BPF program nesting
- Discussion on Probable Solutions and Related Questions
- Summary

Is the kernel stack always in a safe state for a BPF attachment?

- At runtime, Given the limited memory footprint of BPF programs and the controlled state of Kernel stack memory, one can assume that attachments are consistently innocuous.

- Nonetheless, there have been cases reported in the file systems and networking communities where a significant amount of kernel stack memory was used in certain scenarios.

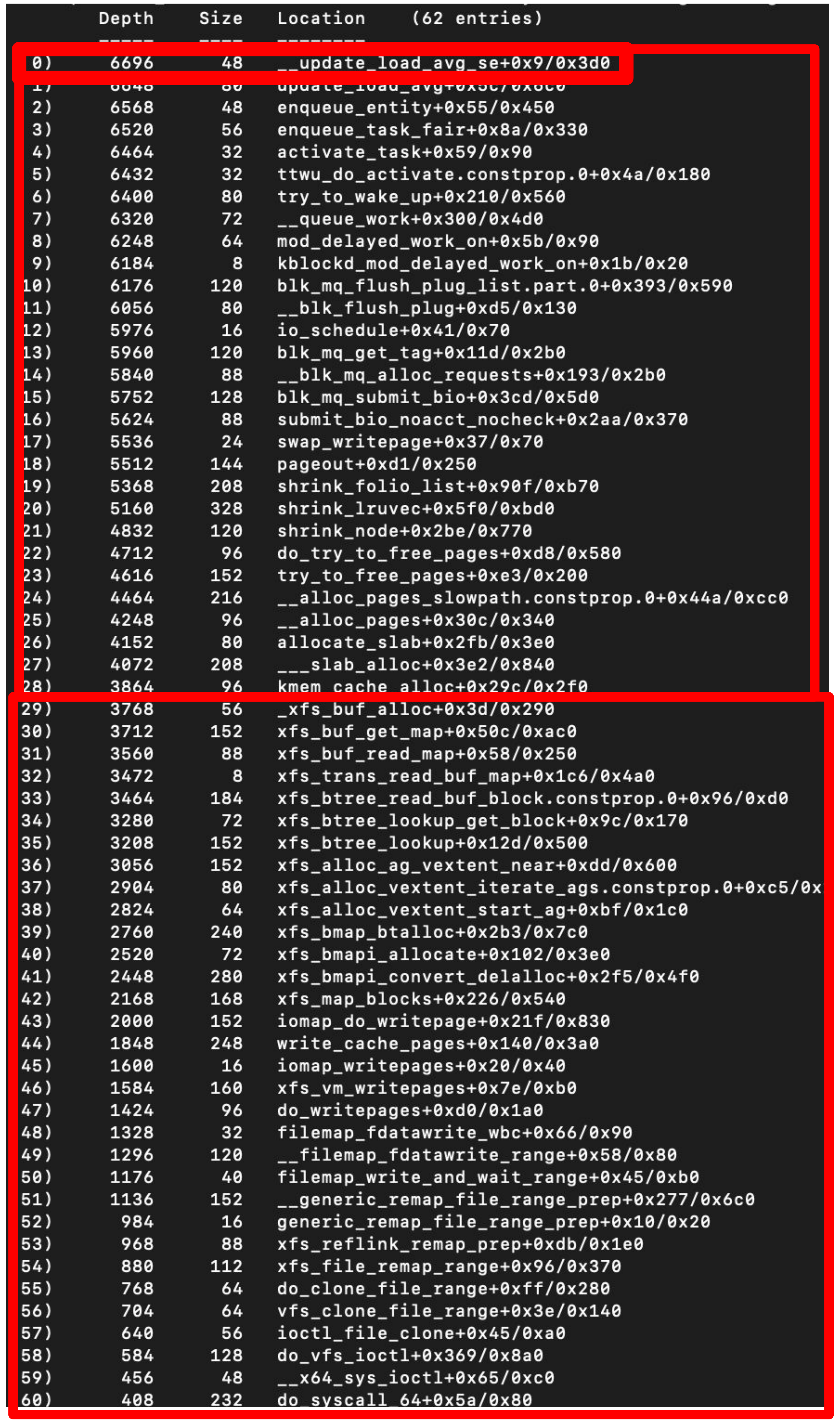
- What if a BPF program is attached on such a kernel stack state?



High stack usage using XFS file system

- To test our assumption, we considered XFS filesystem and ran a XFS test under certain memory constrained situation. It created >6KB of base stack.

- System configuration: Intel x86_64 running on a VM using 1 core, 258 KB memory, 2GB swap memory.

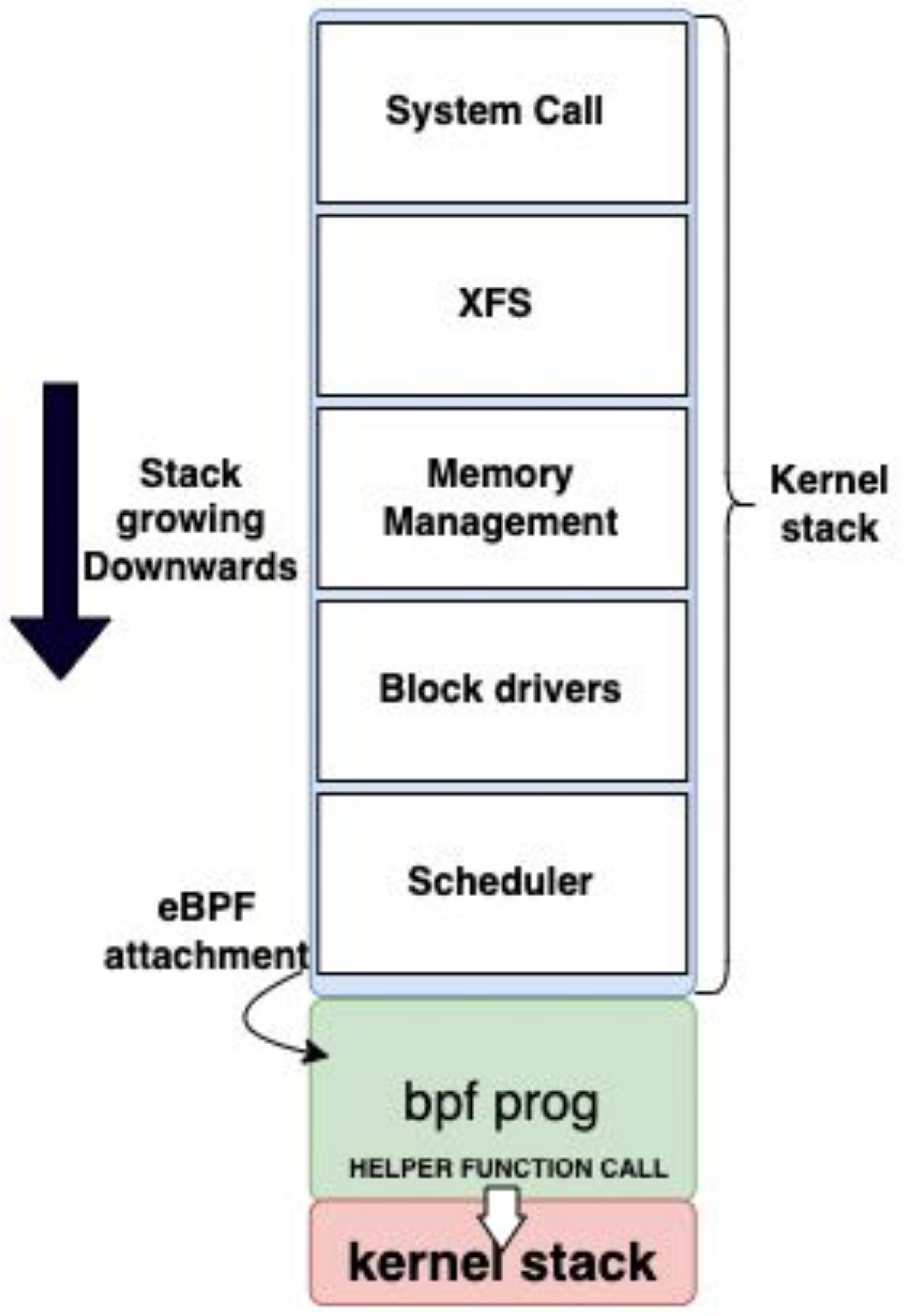


```
        Depth    Size   Location    (62 entries)
        -----    ----   --------
  0)     6696      48   __update_load_avg_se+0x9/0x3d0
  1)     6648      80   update_load_avg+0x5c/0x6c0
  2)     6568      48   enqueue_entity+0x55/0x450
  3)     6520      56   enqueue_task_fair+0x8a/0x330
  4)     6464      32   activate_task+0x59/0x90
  5)     6432      32   ttwu_do_activate.constprop.0+0x4a/0x180
  6)     6400      80   try_to_wake_up+0x210/0x560
  7)     6320      72   __queue_work+0x300/0x4d0
  8)     6248      64   mod_delayed_work_on+0x5b/0x90
  9)     6184       8   kblockd_mod_delayed_work_on+0x1b/0x20
 10)     6176     120   blk_mq_flush_plug_list.part.0+0x393/0x590
 11)     6056      80   __blk_flush_plug+0xd5/0x130
 12)     5976      16   io_schedule+0x41/0x70
 13)     5960     120   blk_mq_get_tag+0x11d/0x2b0
 14)     5840      88   __blk_mq_alloc_requests+0x193/0x2b0
 15)     5752     128   blk_mq_submit_bio+0x3cd/0x5d0
 16)     5624      88   submit_bio_noacct_nocheck+0x2aa/0x370
 17)     5536      24   swap_writepage+0x37/0x70
 18)     5512     144   pageout+0xd1/0x250
 19)     5368     208   shrink_folio_list+0x90f/0xb70
 20)     5160     328   shrink_lruvec+0x5f0/0xbd0
 21)     4832     120   shrink_node+0x2be/0x770
 22)     4712      96   do_try_to_free_pages+0xd8/0x580
 23)     4616     152   try_to_free_pages+0xe3/0x200
 24)     4464     216   __alloc_pages_slowpath.constprop.0+0x44a/0xcc0
 25)     4248      96   __alloc_pages+0x30c/0x340
 26)     4152      80   allocate_slab+0x2fb/0x3e0
 27)     4072     208   ___slab_alloc+0x3e2/0x840
 28)     3864      96   kmem_cache_alloc+0x29c/0x2f0
 29)     3768      56   _xfs_buf_alloc+0x3d/0x290
 30)     3712     152   xfs_buf_get_map+0x50c/0xac0
 31)     3560      88   xfs_buf_read_map+0x58/0x250
 32)     3472       8   xfs_trans_read_buf_map+0x1c6/0x4a0
 33)     3464     184   xfs_btree_read_buf_block.constprop.0+0x96/0xd0
 34)     3280      72   xfs_btree_lookup_get_block+0x9c/0x170
 35)     3208     152   xfs_btree_lookup+0x12d/0x500
 36)     3056     152   xfs_alloc_ag_vextent_near+0xdd/0x600
 37)     2904      80   xfs_alloc_vextent_iterate_ags.constprop.0+0xc5/0x
 38)     2824      64   xfs_alloc_vextent_start_ag+0xbf/0x1c0
 39)     2760     240   xfs_bmap_btalloc+0x2b3/0x7c0
 40)     2520      72   xfs_bmapi_allocate+0x102/0x3e0
 41)     2448     280   xfs_bmapi_convert_delalloc+0x2f5/0x4f0
 42)     2168     168   xfs_map_blocks+0x226/0x540
 43)     2000     152   iomap_do_writepage+0x21f/0x830
 44)     1848     248   write_cache_pages+0x140/0x3a0
 45)     1600      16   iomap_writepages+0x20/0x40
 46)     1584     160   xfs_vm_writepages+0x7e/0xb0
 47)     1424      96   do_writepages+0xd0/0x1a0
 48)     1328      32   filemap_fdatawrite_wbc+0x66/0x90
 49)     1296     120   __filemap_fdatawrite_range+0x58/0x80
 50)     1176      40   filemap_write_and_wait_range+0x45/0xb0
 51)     1136     152   __generic_remap_file_range_prep+0x277/0x6c0
 52)      984      16   generic_remap_file_range_prep+0x10/0x20
 53)      968      88   xfs_reflink_remap_prep+0xdb/0x1e0
 54)      880     112   xfs_file_remap_range+0x96/0x370
 55)      768      64   do_clone_file_range+0xff/0x280
 56)      704      64   vfs_clone_file_range+0x3e/0x140
 57)      640      56   ioctl_file_clone+0x45/0xa0
 58)      584     128   do_vfs_ioctl+0x369/0x8a0
 59)      456      48   __x64_sys_ioctl+0x65/0xc0
 60)      408     232   do_syscall_64+0x5a/0x80
```

Choosing an attachment function

- The most deepest functions of choice at the stack's top in XFS runs were associated with memory management and scheduling tasks, such as list_lru_add() and update_load_avg().

- BPF dynamic attach mechanisms inheriting the bloated stack:
    - When dynamic tracing is active, kprobe optimizes by employing the same kernel stack instead of initiating a new interrupt stack.
    - fentry which uses bpf trampoline by design runs on the same kernel stack using dynamic tracing.

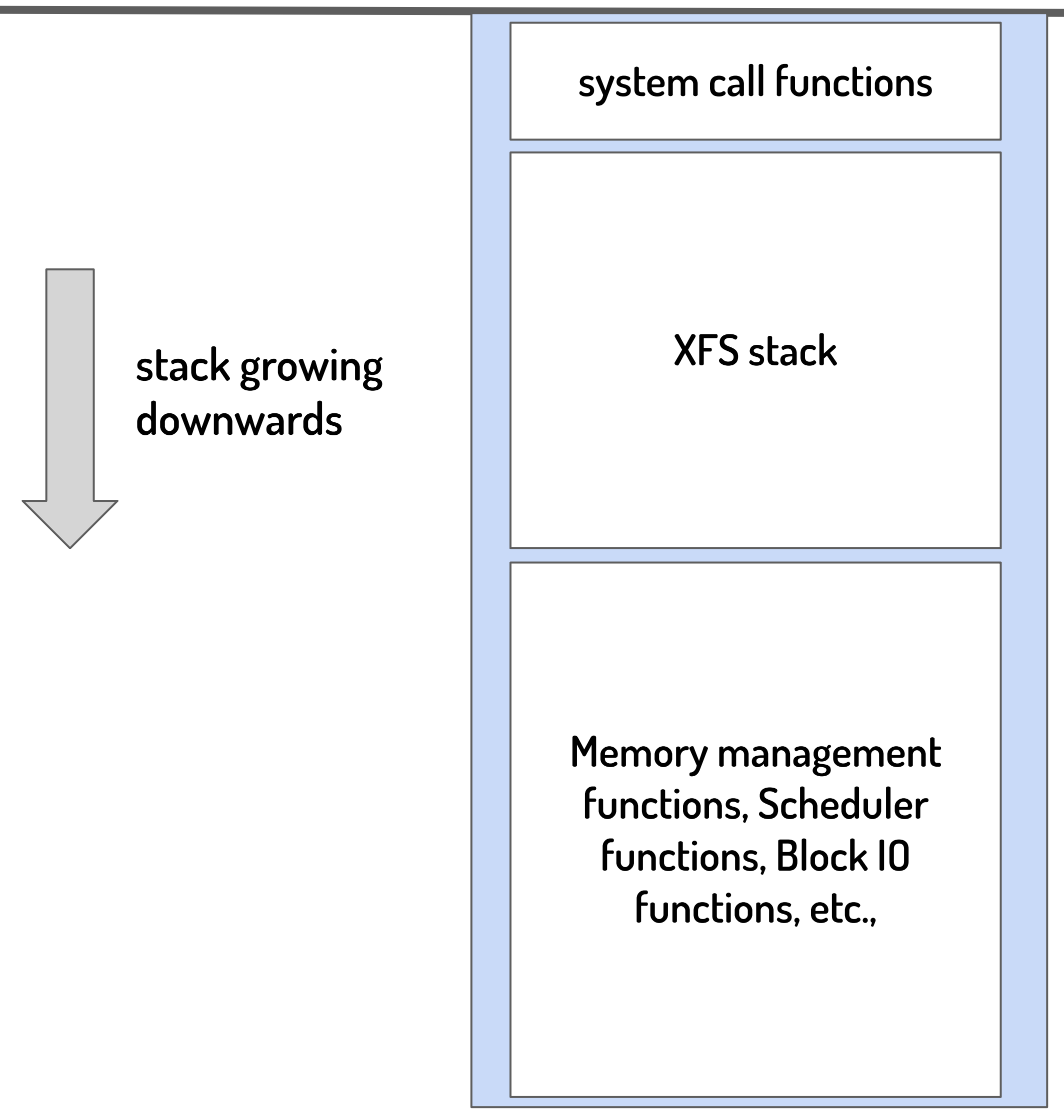


Design of the BPF program

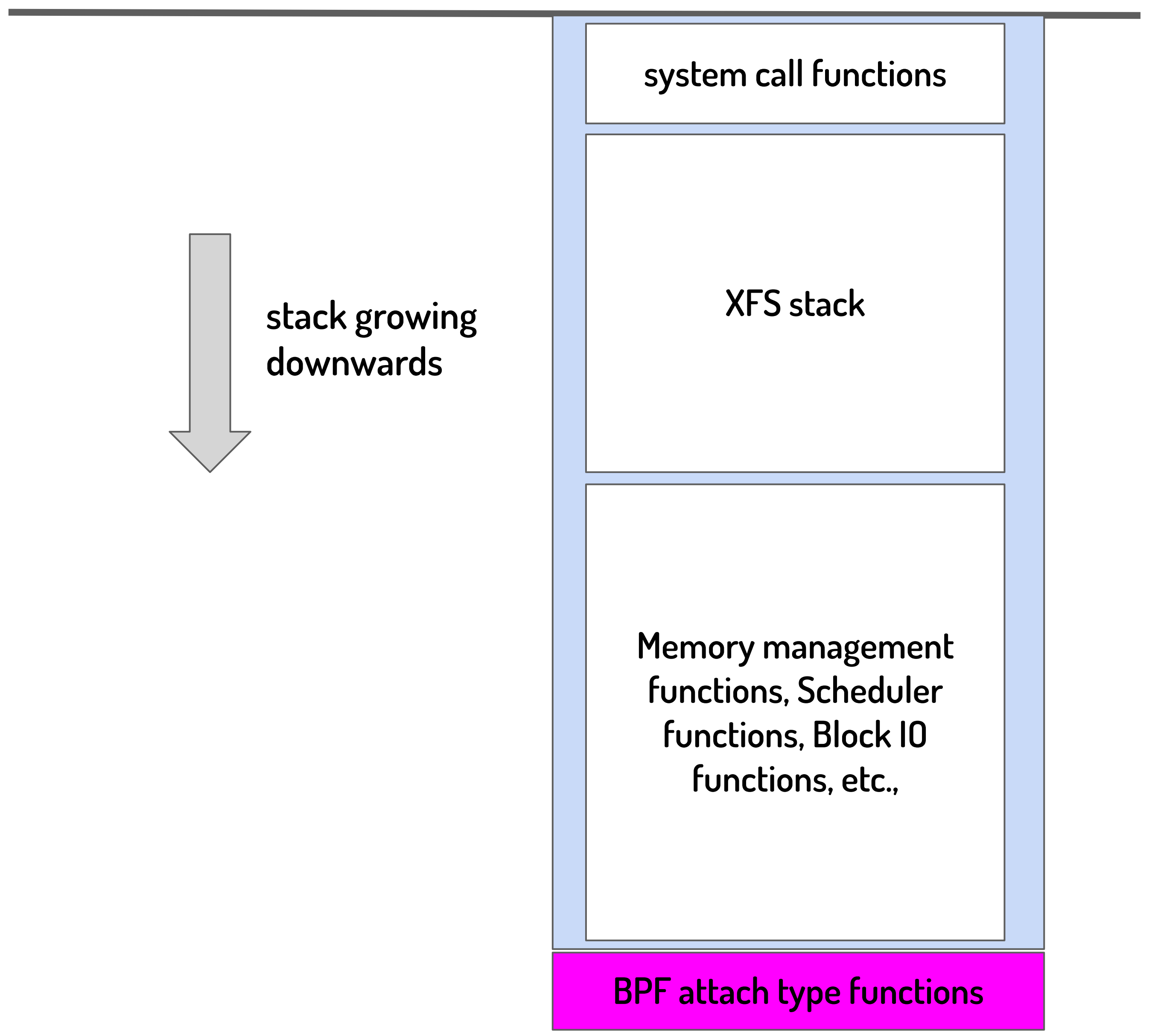
- A BPF program is crafted as previously mentioned, utilizing 33 tail calls, resulting in more than **8KB** of stack usage.
- **Helper functions**, which are not traced by either the verifier or the runtime environment, contribute to further stack growth on top of the BPF program.
- bpf_get_stackid( ) helper function is used in our case, which is called from the top/last tail call bpf program adding more stack space

```
        Depth    Size   Location    (75 entries)
 0)     12160      48   __rcu_read_lock+0x9/0x30
 1)     12112      32   __is_insn_slot_addr+0x18/0x70
 2)     12080      32   kernel_text_address+0x59/0xd0
 3)     12048      16   __kernel_text_address+0x12/0x40
 4)     12032      16   unwind_get_return_address+0x1f/0x30
 5)     12016     152   perf_callchain_kernel+0x9b/0x120
 6)     11864     104   get_perf_callchain+0x9d/0x1b0
 8)     11728     240   bpf_prog_821a4523b619acd1_testing_bpf_to_bpf_calls33+0x13
 9)     11488     240   bpf_prog_b7cca20111e3c722_testing_tail_func33+0x123/0x12a
10)     11248     248   bpf_prog_de824844b398a0ab_testing_tail_func32+0x145/0x14c
11)     11000     248   bpf_prog_a65c2b4afad20f75_testing_tail_func31+0x145/0x14c
12)     10752     248   bpf_prog_faa8165de1ba6c92_testing_tail_func30+0x145/0x14c
13)     10504     248   bpf_prog_450c62cc0bef3e94_testing_tail_func29+0x145/0x14c
14)     10256     248   bpf_prog_eb160495fcc5a5ad_testing_tail_func28+0x145/0x14c
15)     10008     248   bpf_prog_534978f0cfbb9fb9_testing_tail_func27+0x145/0x14c
16)      9760     248   bpf_prog_3f78d05d418998a0_testing_tail_func26+0x145/0x14c
17)      9512     248   bpf_prog_008d7a0b677ebab9_testing_tail_func25+0x145/0x14c
18)      9264     248   bpf_prog_5e576f23637118fe_testing_tail_func24+0x145/0x14c
19)      9016     248   bpf_prog_9822d6d971ba0c69_testing_tail_func23+0x145/0x14c
20)      8768     248   bpf_prog_70bc7fef41b1de7b_testing_tail_func22+0x145/0x14c
21)      8520     248   bpf_prog_89570057b57f114d_testing_tail_func21+0x145/0x14c
22)      8272     248   bpf_prog_55ed2edbc7ae988f_testing_tail_func20+0x145/0x14c
23)      8024     248   bpf_prog_c3fc066b2ed1746c_testing_tail_func19+0x145/0x14c
24)      7776     248   bpf_prog_b009738e7e48d28b_testing_tail_func18+0x145/0x14c
25)      7528     248   bpf_prog_df8fa063c72e23b9_testing_tail_func17+0x145/0x14c
26)      7280     248   bpf_prog_5d4cd9108aa3facb_testing_tail_func16+0x145/0x14c
27)      7032     248   bpf_prog_e7a5882970c1de90_testing_tail_func15+0x145/0x14c
28)      6784     248   bpf_prog_a745c69d19dfe0bd_testing_tail_func14+0x145/0x14c
29)      6536     248   bpf_prog_b905a87e322818d2_testing_tail_func13+0x145/0x14c
30)      6288     248   bpf_prog_0bca5ee992c76dc5_testing_tail_func12+0x145/0x14c
31)      6040     248   bpf_prog_ff55c8349e8959f8_testing_tail_func11+0x145/0x14c
32)      5792     248   bpf_prog_b97d69f618cf9477_testing_tail_func10+0x145/0x14c
33)      5544     248   bpf_prog_11d88266e28aa7d3_testing_tail_func9+0x145/0x14c
34)      5296     248   bpf_prog_d73278b232b243d9_testing_tail_func8+0x145/0x14c
35)      5048     248   bpf_prog_a8777800ada79293_testing_tail_func7+0x145/0x14c
36)      4800     248   bpf_prog_5bc7c1767f4aac84_testing_tail_func6+0x145/0x14c
37)      4552     248   bpf_prog_0fea45fd942a1f7f_testing_tail_func5+0x145/0x14c
38)      4304     248   bpf_prog_53e72671ffa171e5_testing_tail_func4+0x145/0x14c
39)      4056     248   bpf_prog_acd4a4a207e328e8_testing_tail_func3+0x145/0x14c
40)      3808     248   bpf_prog_880dc8647dcd1e01_testing_tail_func2+0x145/0x14c
41)      3560     248   bpf_prog_5137f93fcd07411c_testing_tail_func+0x140/0x147
42)      3312      80   trace_call_bpf+0x9a/0x140
43)      3232     120   kprobe_perf_func+0x53/0x260
44)      3112      56   kprobe_ftrace_handler+0x152/0x200
45)      3056     104   arch_ftrace_ops_list_func+0x154/0x230
46)      2952     184   ftrace_regs_call+0x5/0x52
47)      2768       8   __update_load_avg_se+0x9/0x3d0
```

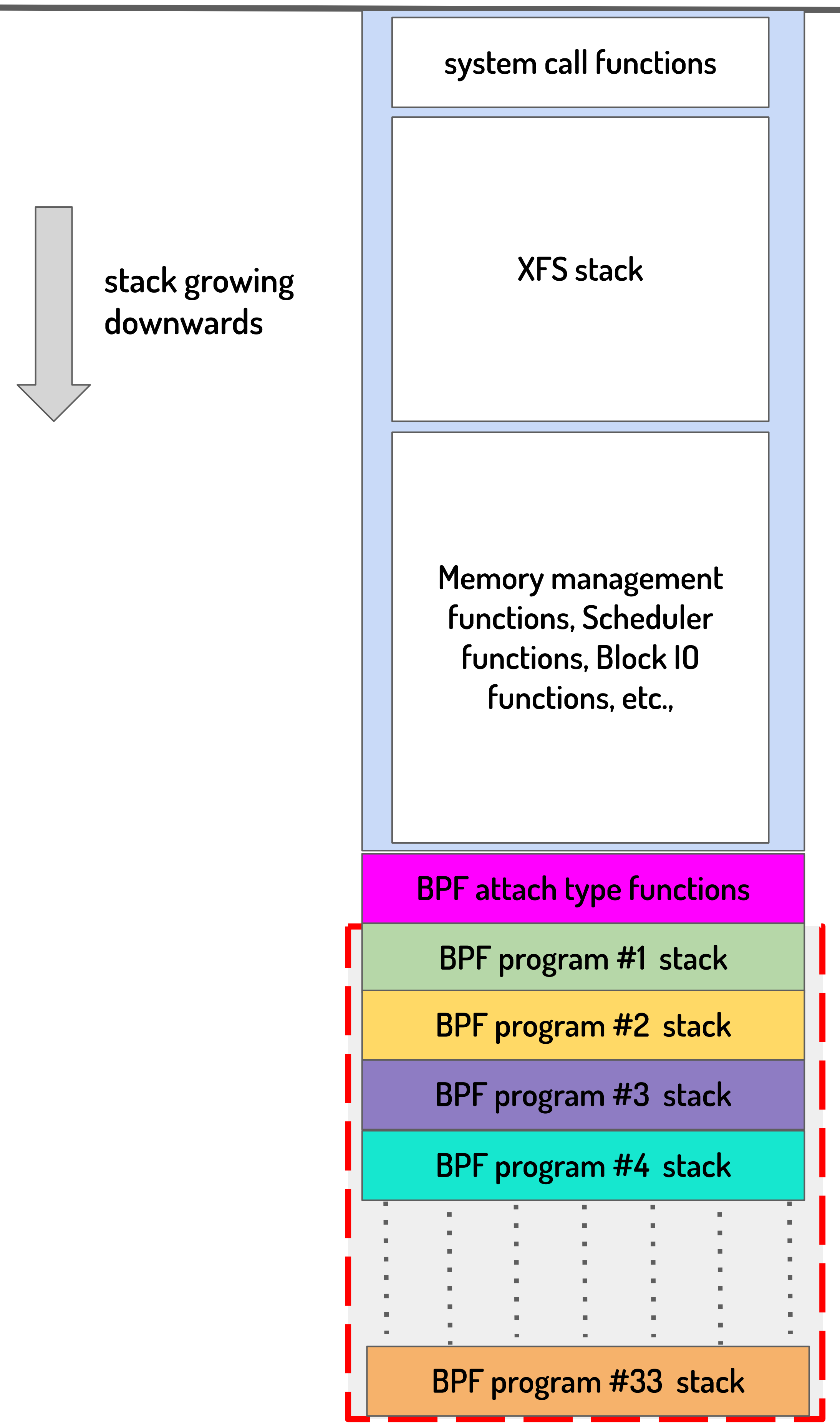
Overflowing a kernel stack using BPF program



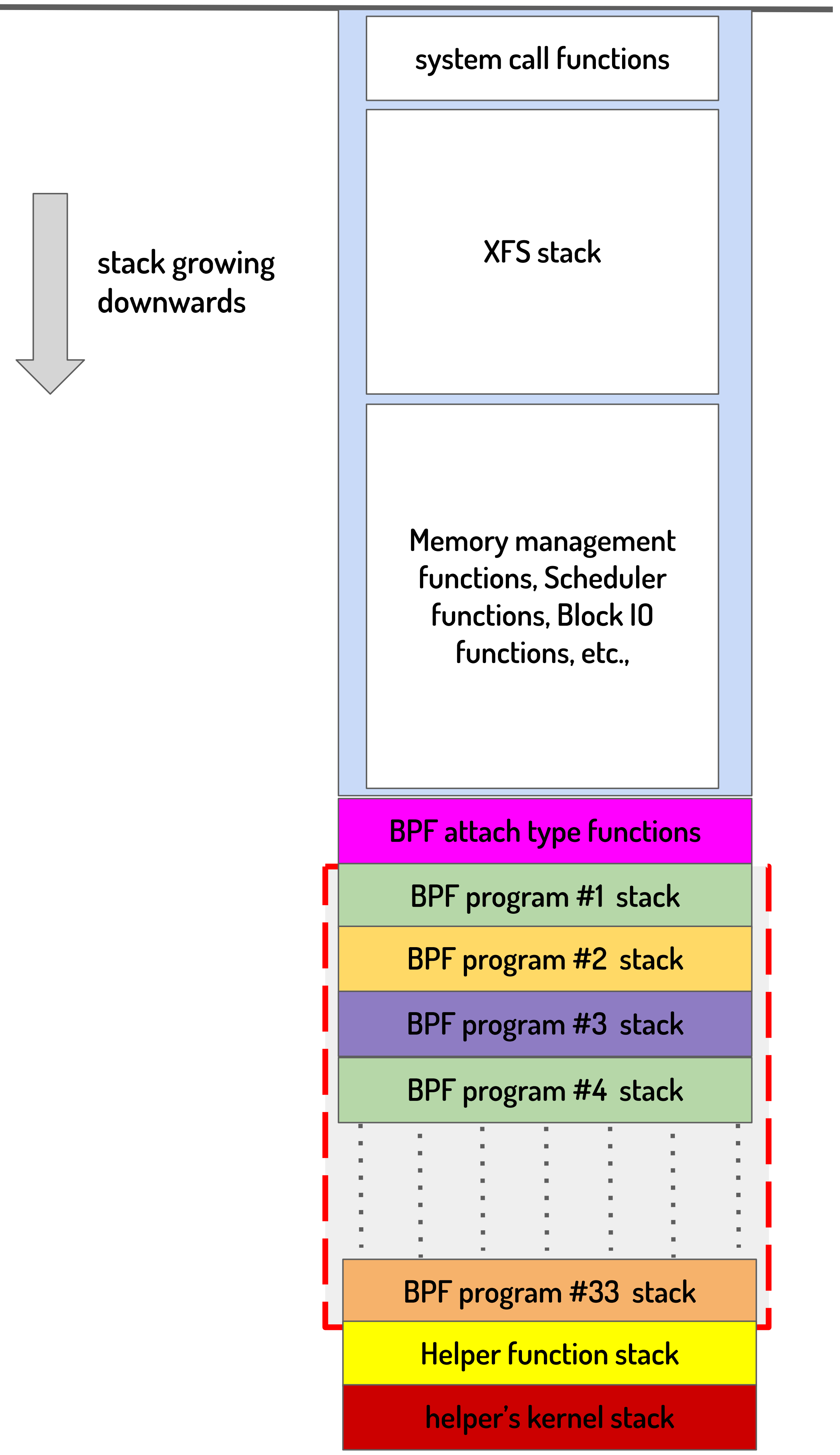
Overflowing a kernel stack using BPF program



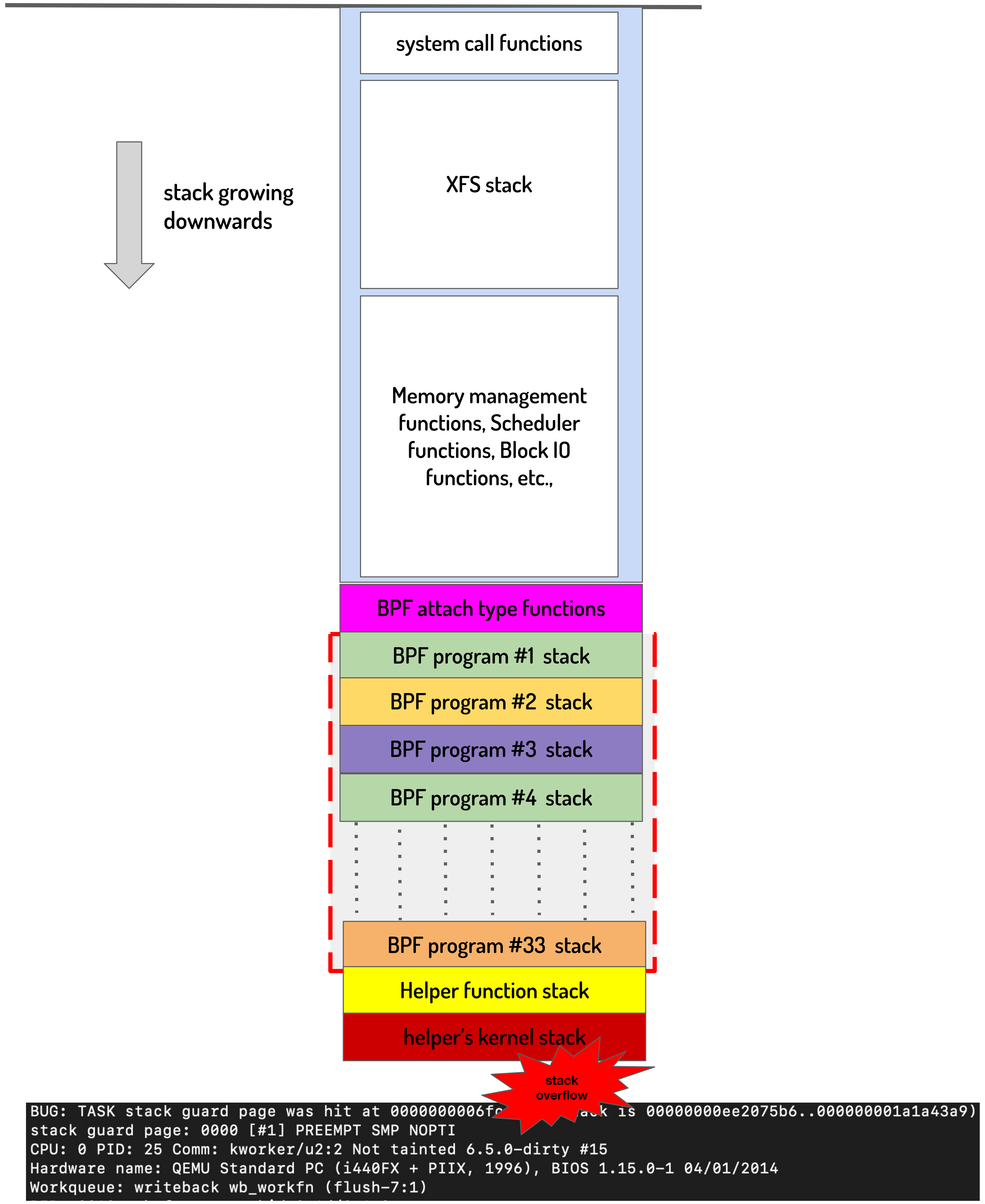
Overflowing a kernel stack using BPF program



Overflowing a kernel stack using BPF program



Overflowing a kernel stack using BPF program

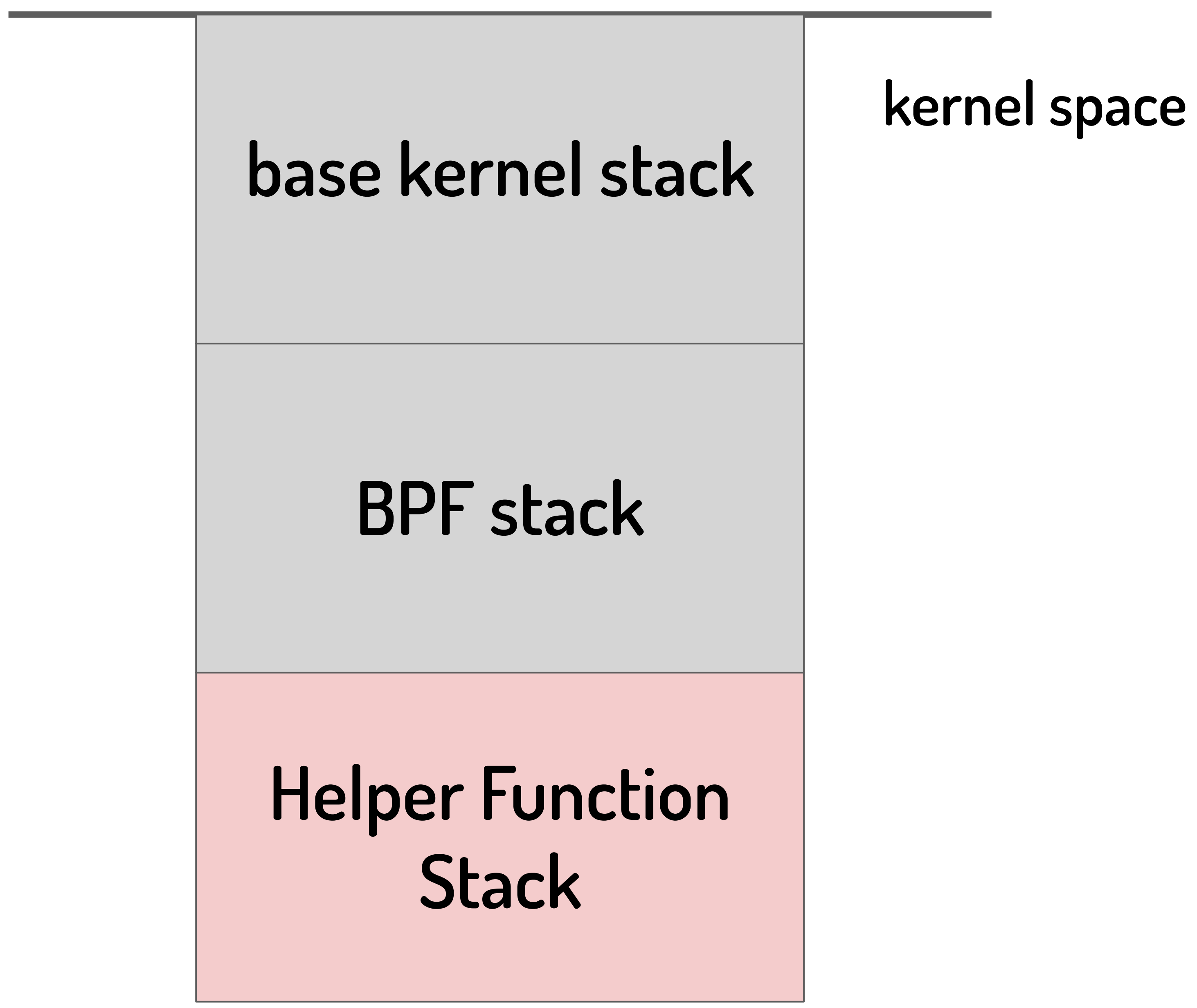


Agenda

- BPF program attachment and its interaction with the stack
- Stack overflow due to BPF program attachment
- **Stack overflow due to uncontrolled BPF program nesting**
- Discussion on Probable Solutions and Related Questions
- Summary

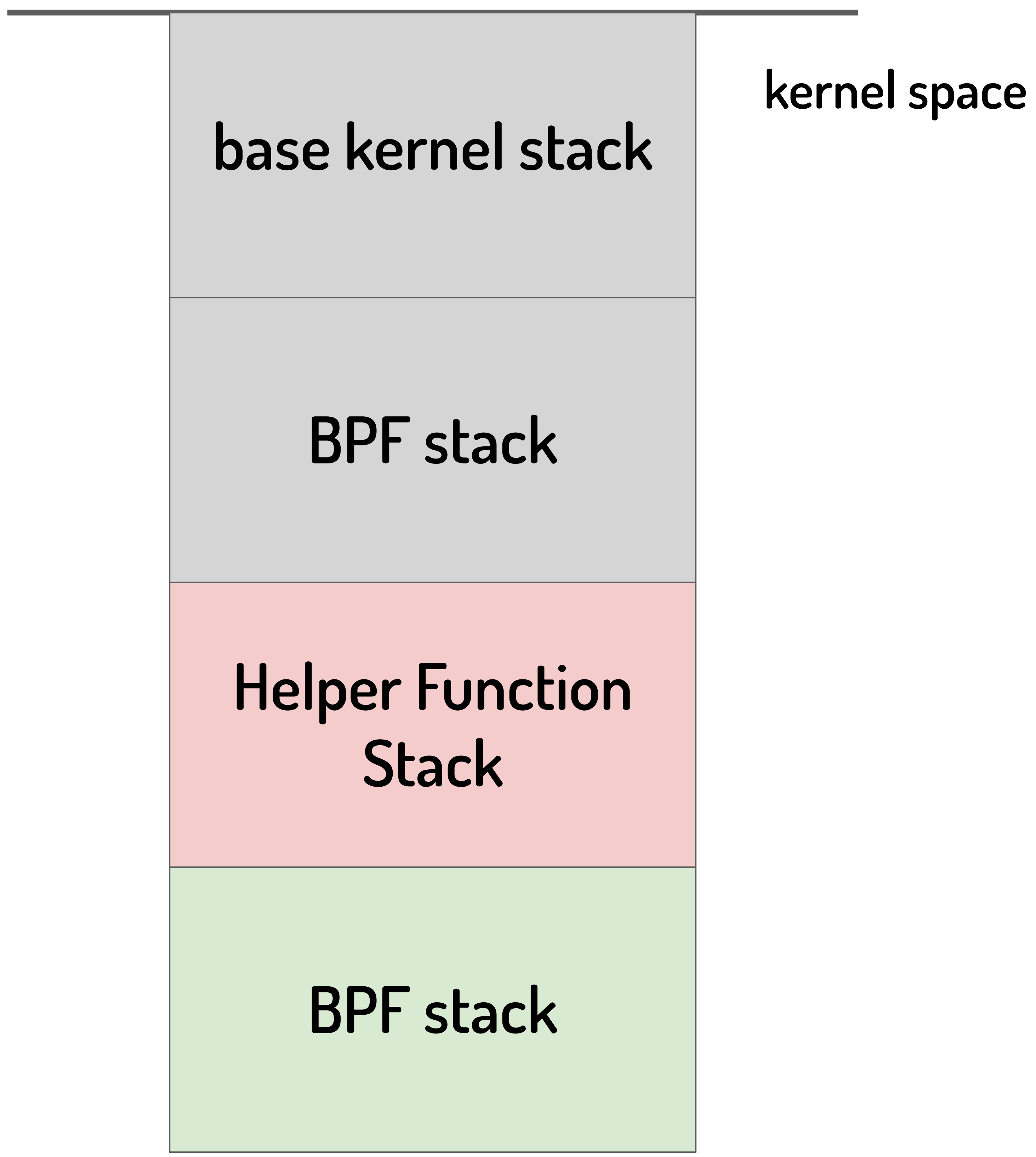
BPF verifier assumptions about BPF programs nesting

- Verifier assumes that helper call stack usage is small.



BPF verifier assumptions about BPF programs nesting

- Verifier assumes that helper call stack usage is small.
- The new desire to nest multiple BPF programs is violating verifier's assumption.



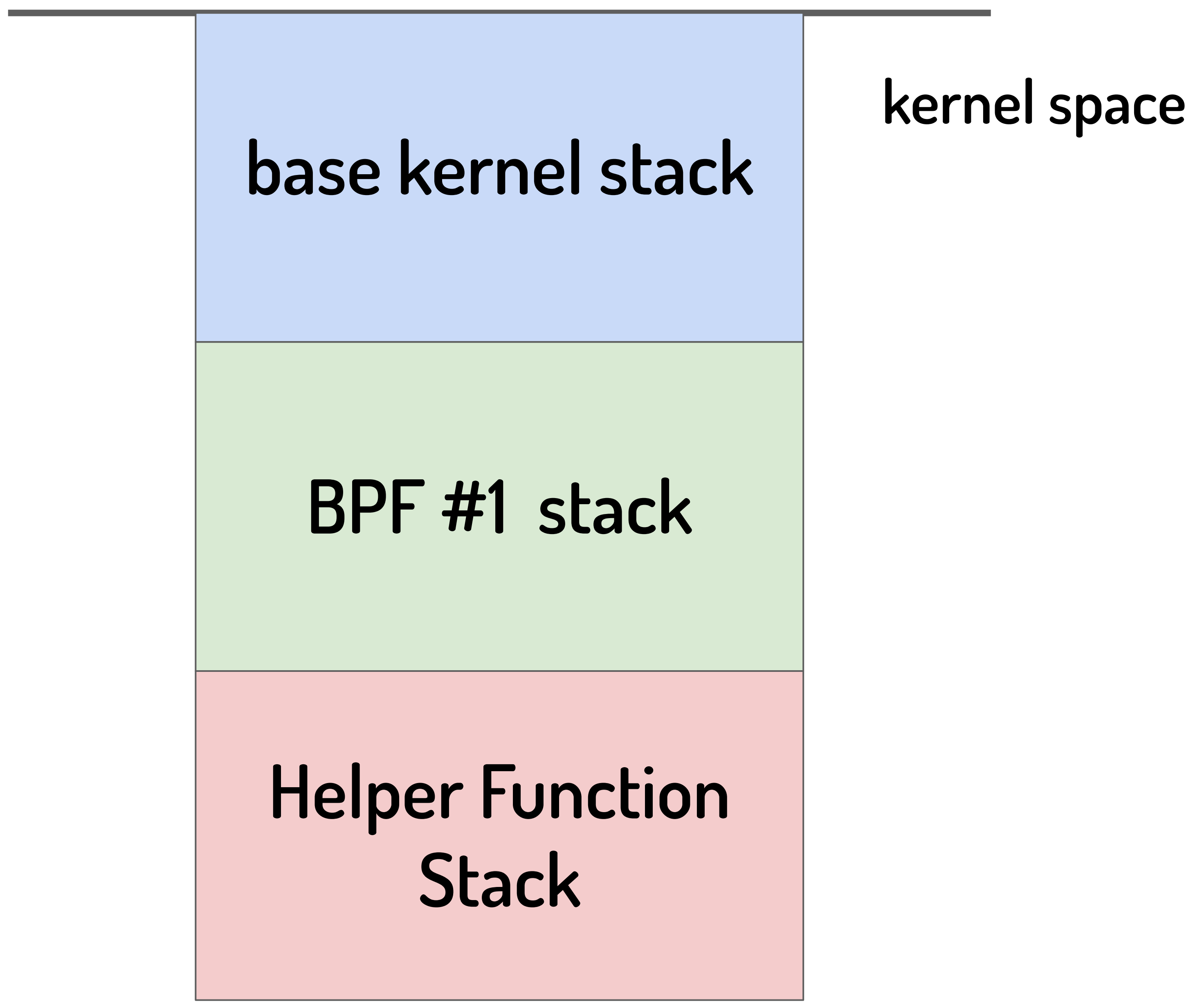
Tracepoints and Kprobe nesting checks for BPF programs

- Tracepoints/ Kprobes doesn't allow nesting of multiple BPF programs.
- Inside `trace_call_bpf` function kernel checks if there is any active BPF program already executing on the same CPU.
- If this condition is true, the corresponding BPF program will not be executed.

```
unsigned int trace_call_bpf(struct trace_event_call *call,
void *ctx)
{
    ....
    if (unlikely(__this_cpu_inc_return(bpf_prog_active) != 1)) {
        ret = 0;
        goto out;
    }
    .....
}
```

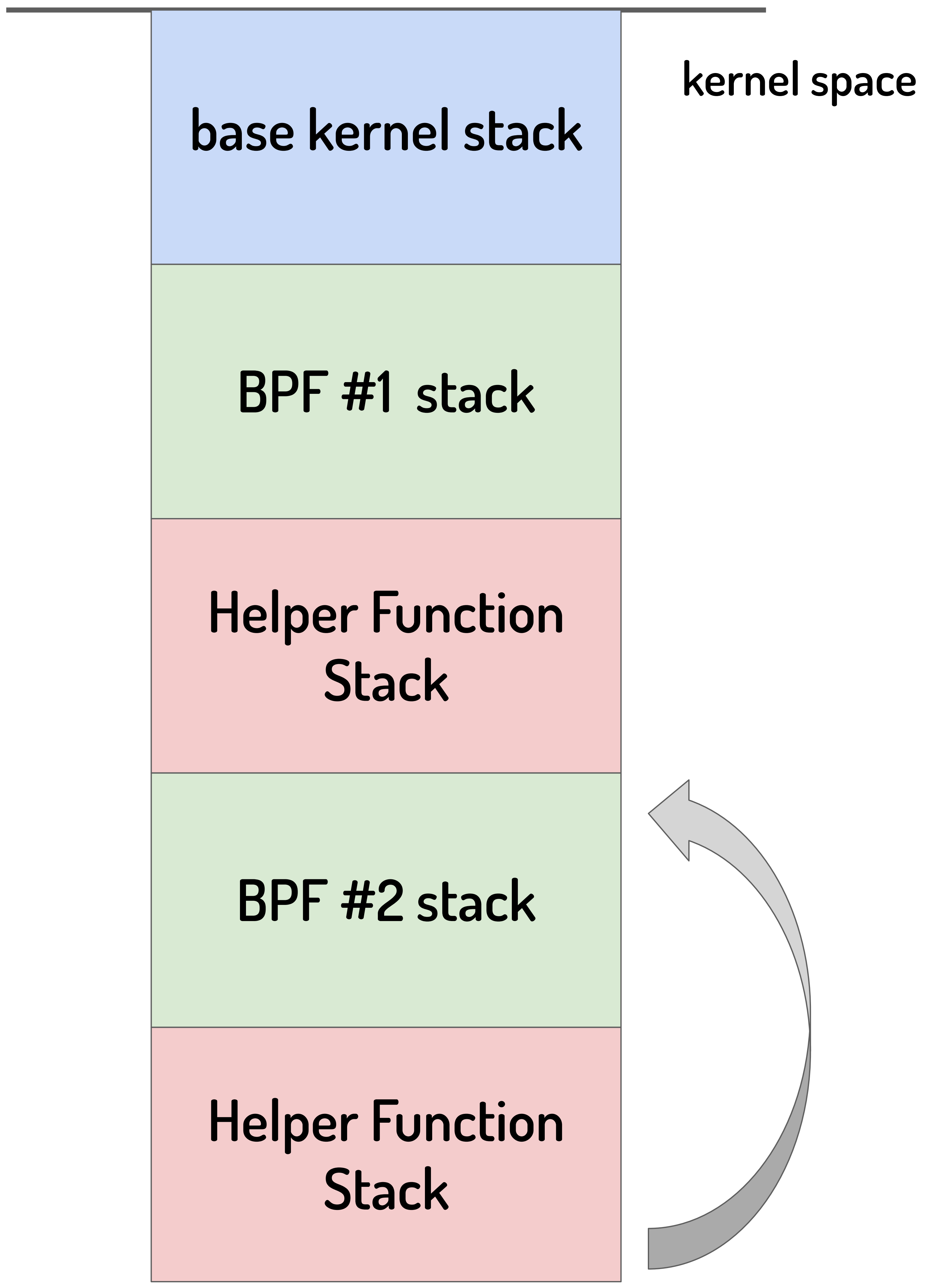
What happens if the nesting checks are not implemented?

- If a BPF program calls a helper function



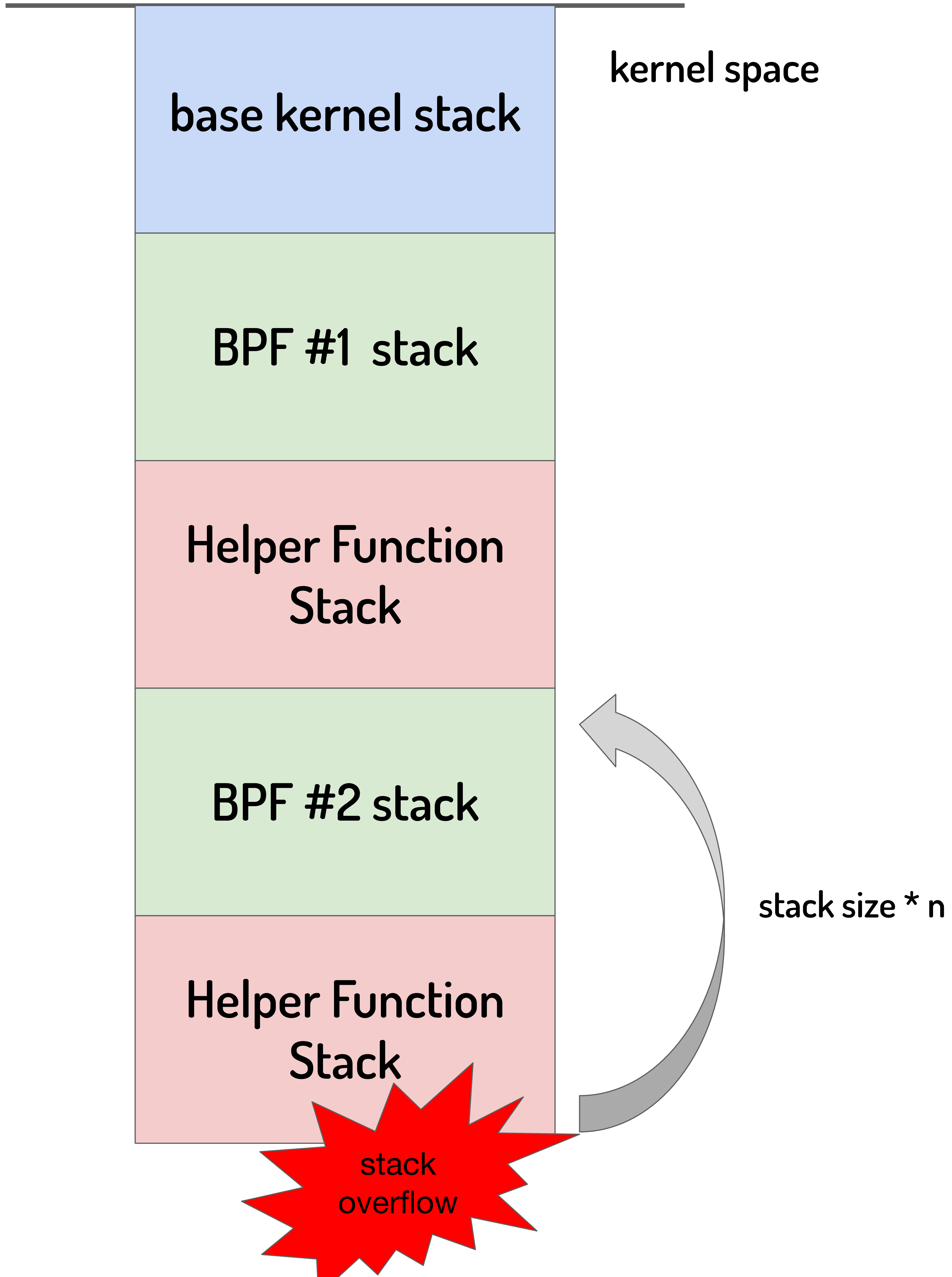
What happens if the nesting checks are not implemented?

- If a BPF program calls a helper function
- and another BPF program, attached to that helper function, calls the same helper, it can create an endless loop.



What happens if the nesting checks are not implemented?

- If a BPF program calls a helper function
- and another BPF program, attached to that helper function, calls the same helper, it can create an endless loop.
- This loop could cause the BPF program to run indefinitely, potentially leading to a system crash due to inheriting the same stack.



Limitations of kprobes/ tracepoints approach

- However, when a BPF program is attached to a helper function or a function invoked within a helper function, the tracing events related to these interactions may not be captured.

What about fentry or trampoline nesting checks attachments?

- BPF Trampoline programs call __bpf_prog_enter_recur before executing BPF instructions. In this function, it essentially checks whether the <u>same</u> BPF program is currently executing on the CPU.

```
static u64 notrace __bpf_prog_enter_recur(struct bpf_prog *prog,
struct bpf_tramp_run_ctx *run_ctx)
	__acquires(RCU)
{
	....
	if (unlikely(this_cpu_inc_return(*(prog->active)) != 1)) {
		bpf_prog_inc_misses_counter(prog);
		return 0;
	}
	....
}
```

Limitations of BPF trampoline approach

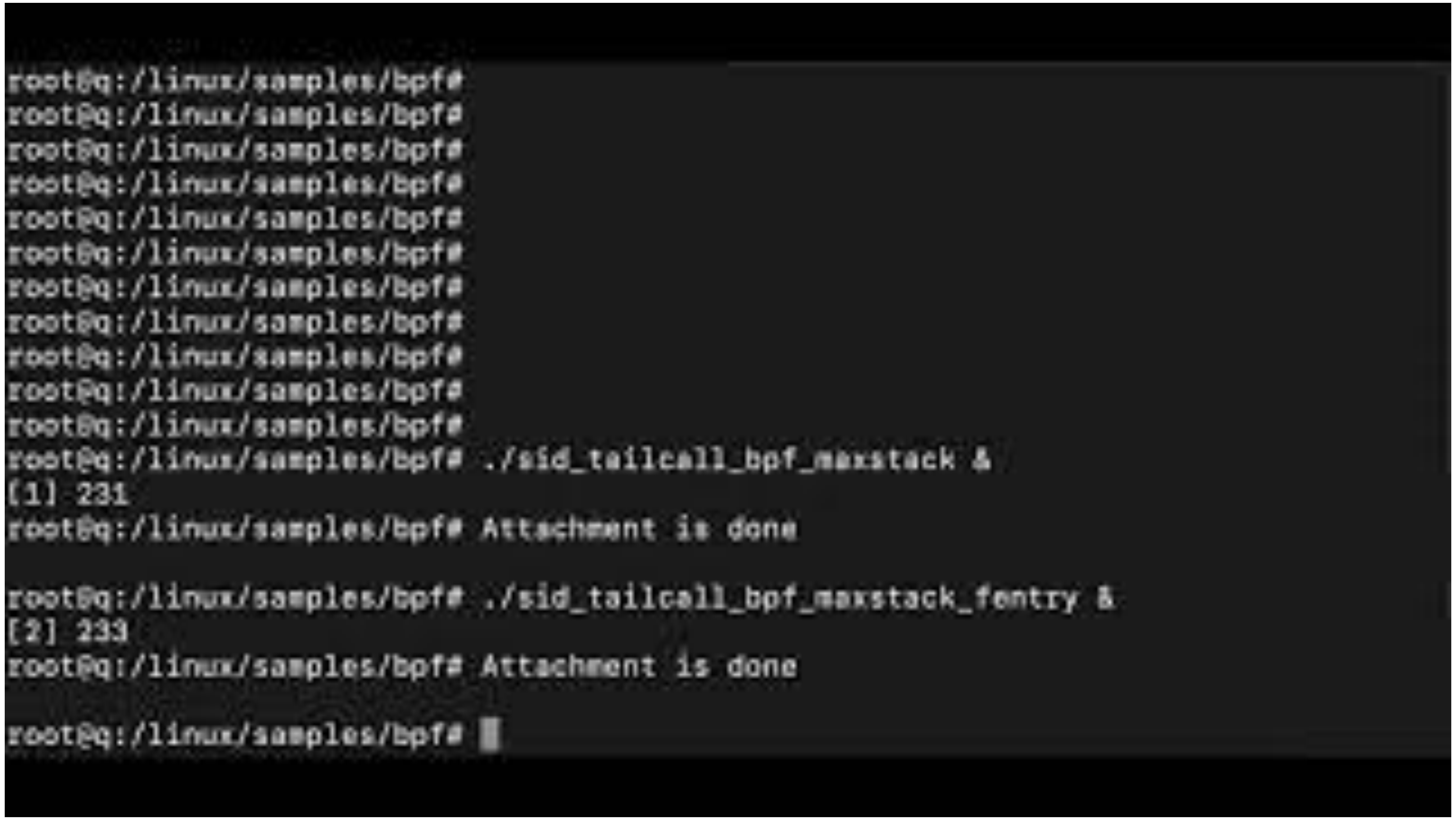
- More than one BPF programs can run on a same CPU, which results in using the same stack.
- So by using nesting multiple BPF programs we can overflow the kernel stack.

Nesting BPF trampoline to overflow stack with other attachments

- In this test, an 8KB BPF program stack, akin to the previous one, is created and attached via a kprobe on the __sys_socket() function

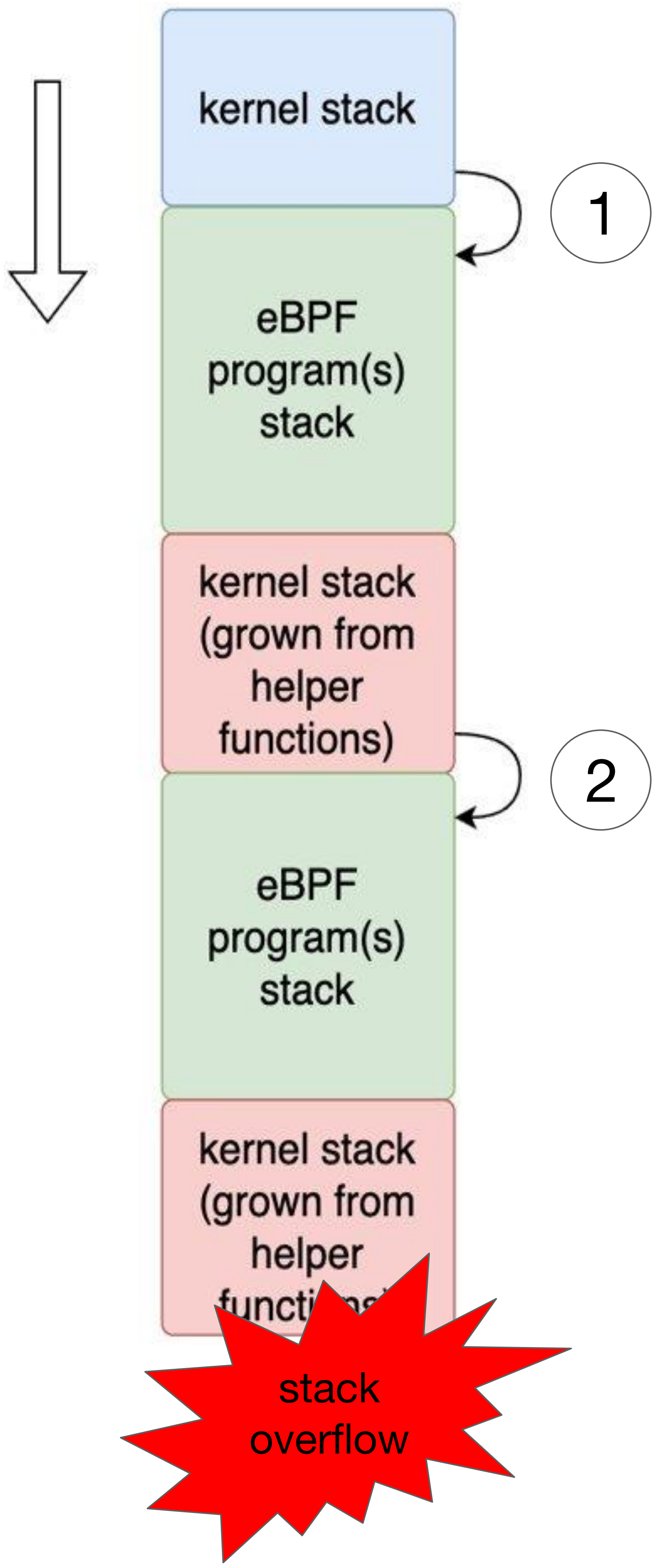
- Adding another 8KB BPF stack through a bpf trampoline to the bpf_get_stackid() helper function or its kernel path leads to an overflow of the x86_64 Linux kernel stack.

Demo Video showing the stackoverflow



Note: Running Linux version v.6.5.0 using QEMU with DYNAMIC_TRACING enabled.

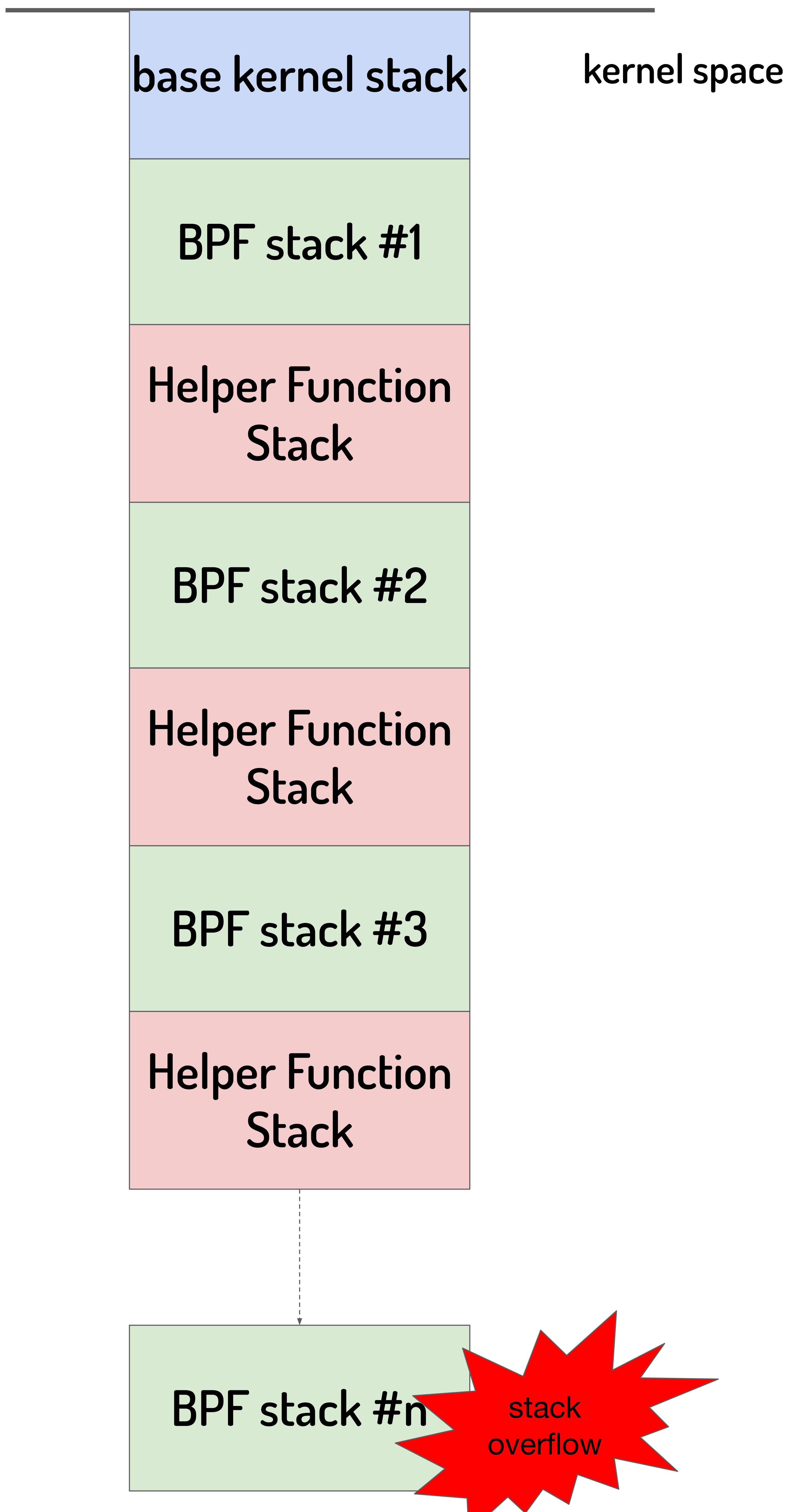
Other possible potential attachments used with BPF trampoline



Attached to hook 1	Attached to hook 2
KPROBE Prog	Fentry Trampoline Prog
Fentry Trampoline Prog	Fentry Trampoline Prog
Tracepoint	Fentry Trampoline Prog
Fentry Trampoline Prog	KPROBE Prog

Overflowing the stack by attaching multiple trampoline BPF programs

- By using BPF programs attached with trampoline without relying on any tail calls and BPF-to-BPF functionality one can overflow the kernel stack.



Agenda

- BPF program attachment and its interaction with the stack
- Stack overflow due to BPF program attachment
- Stack overflow due to uncontrolled BPF program nesting
- **Discussion on Probable Solutions and Related Questions**
- Summary

Requirements to address the problems we discussed

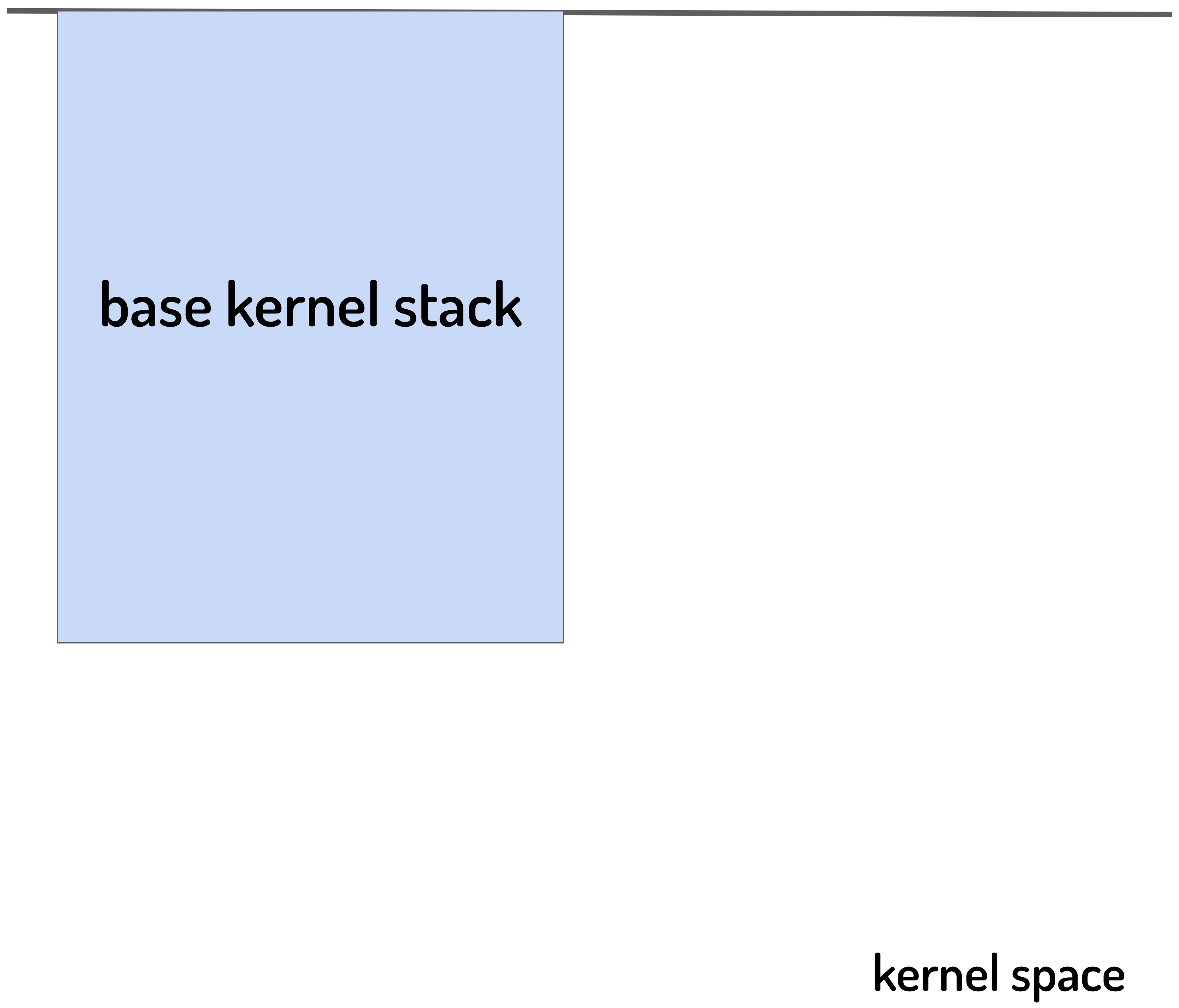
We came up with the following requirements to mitigate the problems caused by the implicit verifier assumptions about the stack state during kernel runtime

1. Kernel runtime should ensure and be able to accommodate the stack space required by the BPF programs to run,

2. and if there is a situation where it cannot allocate enough space for the BPF program, it should prevent it from running.

Probable Solution to Address P#1 Could be: Stack-Switching

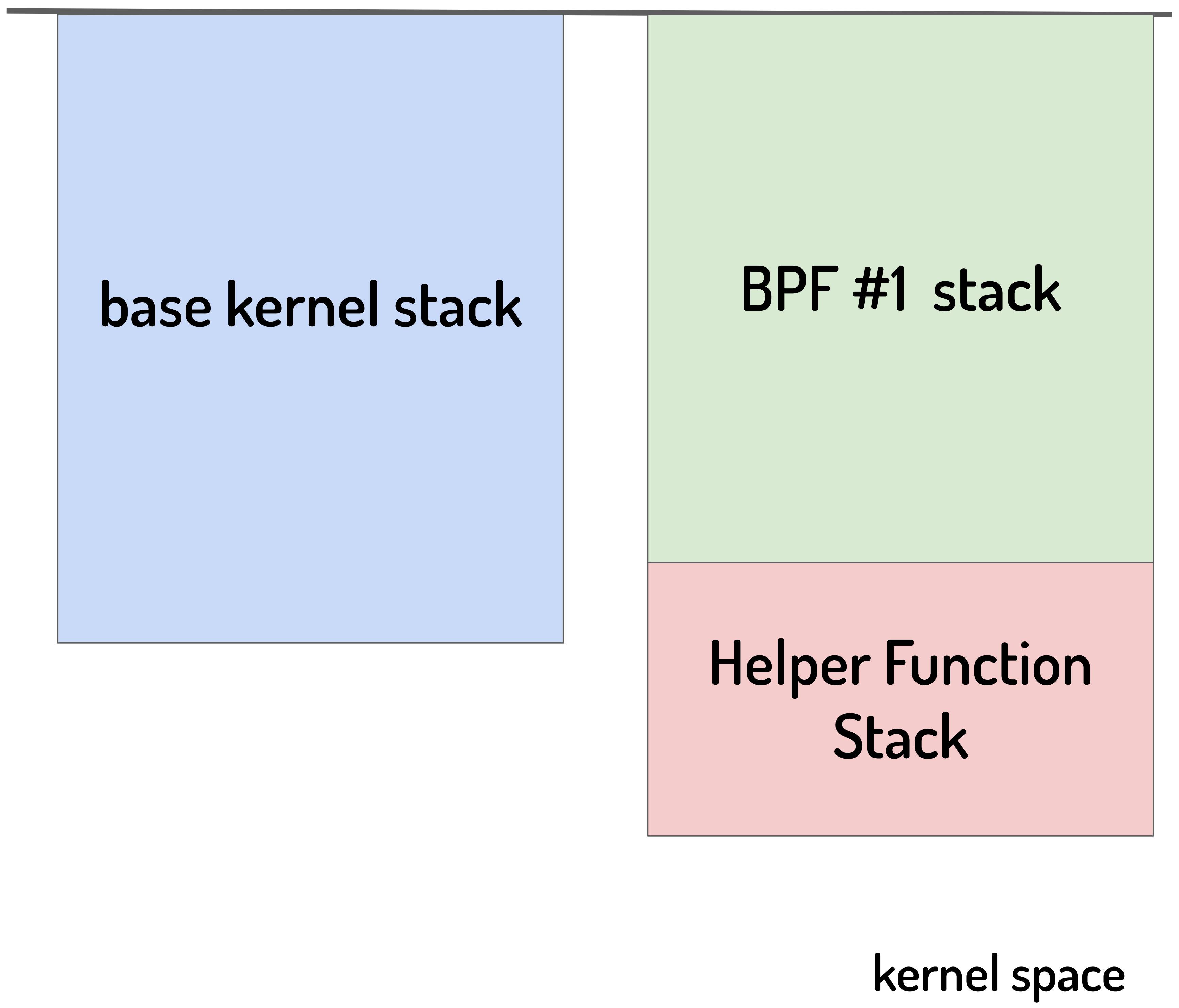
For addressing the issue of stack overflow when attaching a BPF program to an unknown stack state



Probable Solution to Address P#1 Could be: Stack-Switching

For addressing the issue of stack overflow when attaching a BPF program to an unknown stack state

- The kernel can implement a mechanism to switch a BPF program to a new stack based on memory requirements.



Probable Solution to Address P#2 Could be: Stack-Switching and limit nesting

- The stack switching solution can also address the stack overflow issue that occurs with nesting.

- A new stack can be created according to the memory needs of nested programs, and the kernel should enforce a limit on the nesting depth.

Points for Discussion on probable solutions

In these two cases there is a chance that a BPF program might not get executed

1. If we request memory for the stack, there might be a chance that there is not enough space, and the kernel might prevent the BPF program from running.

2. By posing a limit on nesting, a perf BPF program might not get executed.

Open Question: What happens if a BPF program never gets executed

How does this impact critical BPF program extensions designed for purposes like security?

- Examples could be impacts of not running LSM BPF program/ Seccomp filter.

Open Question: Can an orchestration tool solve our problems?

Can BPF orchestrations tools like bpfd alert admins to monitor BPF programs so that we never run into stack overflow problems?

Agenda

- BPF program attachment and its interaction with the stack
- Stack overflow due to BPF program attachment
- Stack overflow due to uncontrolled BPF program nesting
- Discussion on Probable Solutions and Related Questions
- Summary

Summary

- It’s is important to upheld the verifiers assumptions about stack during kernel runtime.
- Violations of these assumptions leads to stackoverflow issues.
- We showed two such cases in our presentation.
    1. Incorrect assumptions about availability of stack state.
    2. Uncontrolled nesting.
- Finally, we raised discussion points on probable solutions to mitigate these issues in the future and raised open questions.

Linux Plumbers Conference

Richmond, Virginia | November 13-15, 2023