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

When BPF programs need to die

Exploring the design space for early BPF termination

Raj Sahu <raj.sahu@vt.edu>
Dan Williams <djwillia@vt.edu>

VIRGINIA TECH

What makes BPF so cool

BPF : **Safe** kernel extensions

Verifier guarantees :

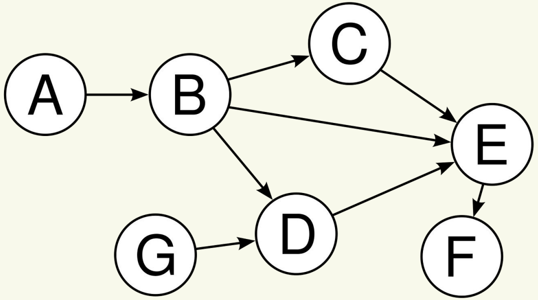
└ Memory Safety :

No {Use-after-Free, Null dereference, Resource leaks}

└ Guaranteed Termination : No {Infinite Loops}

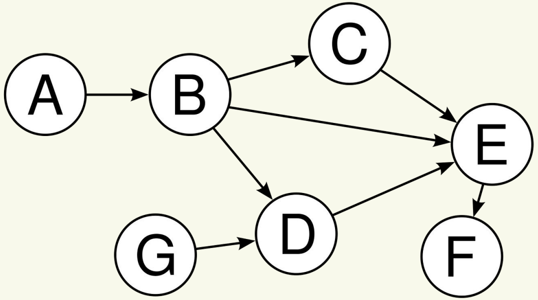
Untrusted code cannot crash (Memory Safety) or stall (Guaranteed Termination) kernel.

Termination as a guarantee



1. Verifier's check on DAG ensures every verifier BPF program will always terminate.

2. Instruction limits, Stack and nesting limits



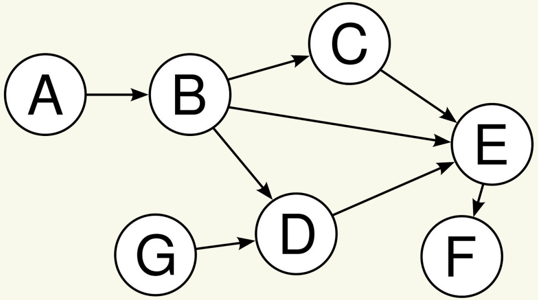
Termination as a guarantee

1. Verifier’s check on DAG ensures every verifier BPF program will always terminate.

2. Instruction limits, Stack and nesting limits

Therefore, a verified BPF program will always terminate in an insignificant time.

Termination as a guarantee



1. Verifier's check on DAG ensures every verifier BPF program will always terminate.

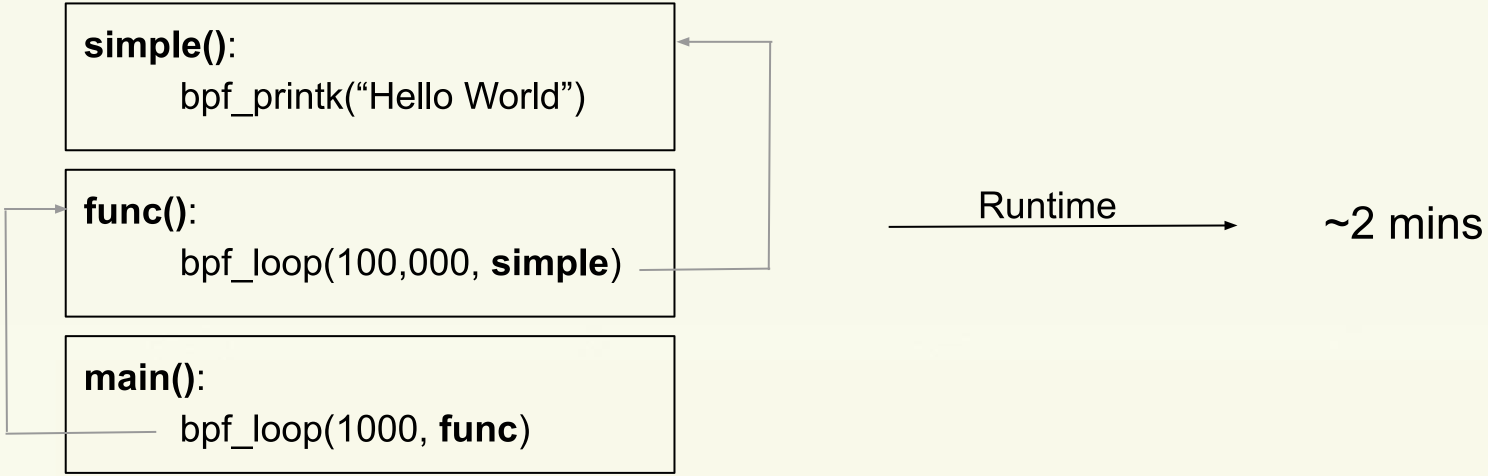
2. Instruction limits, Stack and nesting limits

Therefore, a verified BPF program will always terminate in an ***insignificant*** time.

But some helpers are weakening this guarantee

- bpf_for_each_map_elem
  - Iterates through each element in map and calls a callback function
- bpf_loop
  - Bounded loop on a callback function
- bpf_user_ringbuf_drain
  - Invoke a callback for each sample in a user ring buffer.
- bpf_find_vma
  - maps an address of a task to the vma (vm_area_struct) for this address, and feed the vma to a callback BPF function.

An example long running program



```
simple():
        bpf_printk(“Hello World”)

func():
        bpf_loop(100,000, simple)

main():
        bpf_loop(1000, func)
```

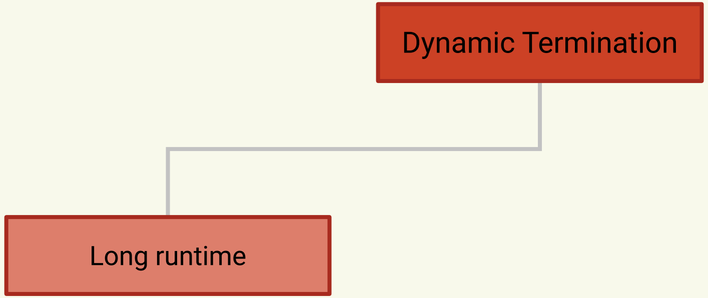
Guaranteed Termination ≠ Fast Termination

We need a Runtime Mechanism !

When do we need Dynamic Termination

Dynamic Termination

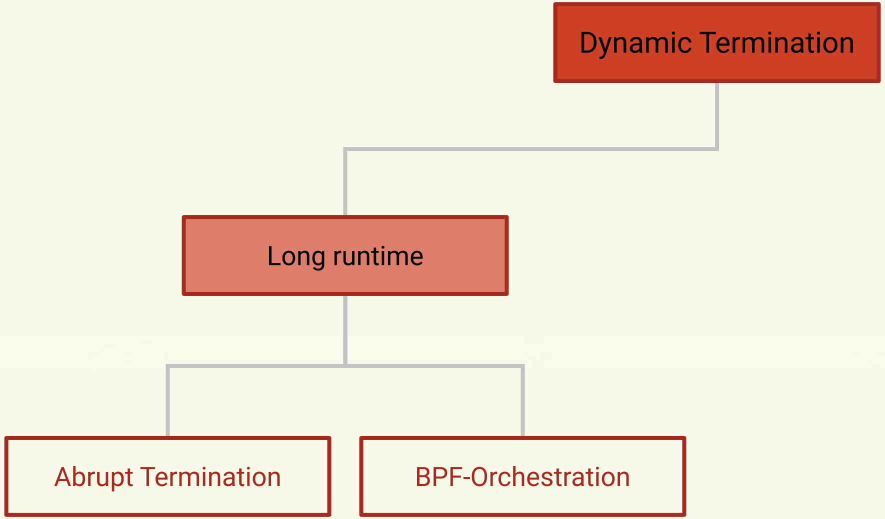
When do we need Dynamic Termination



Dynamic Termination

Long runtime

When do we need Dynamic Termination



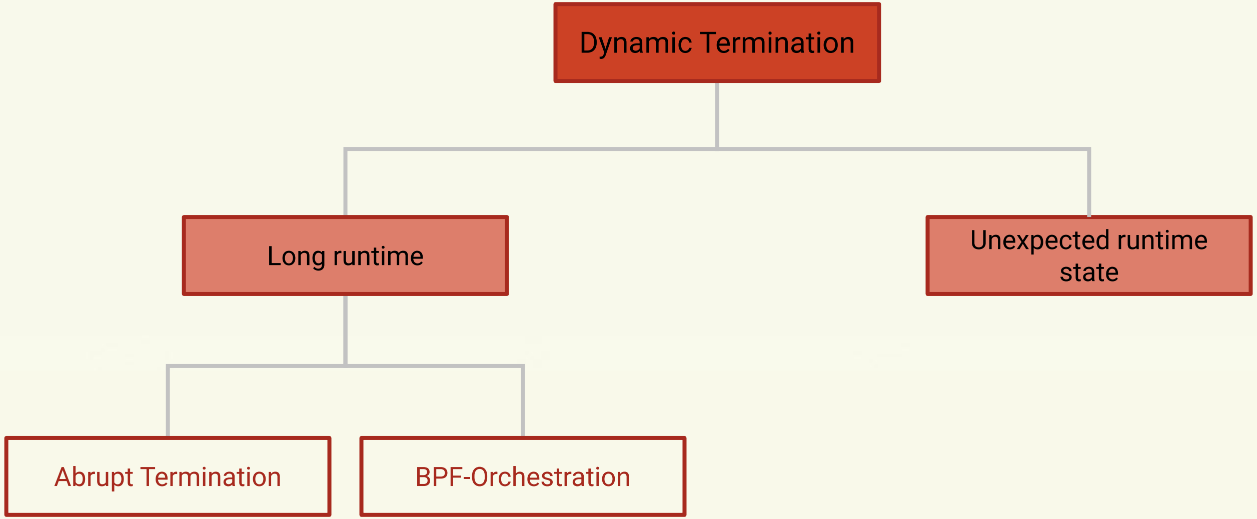
Dynamic Termination

Long runtime

Abrupt Termination

BPF-Orchestration

When do we need Dynamic Termination



Dynamic Termination

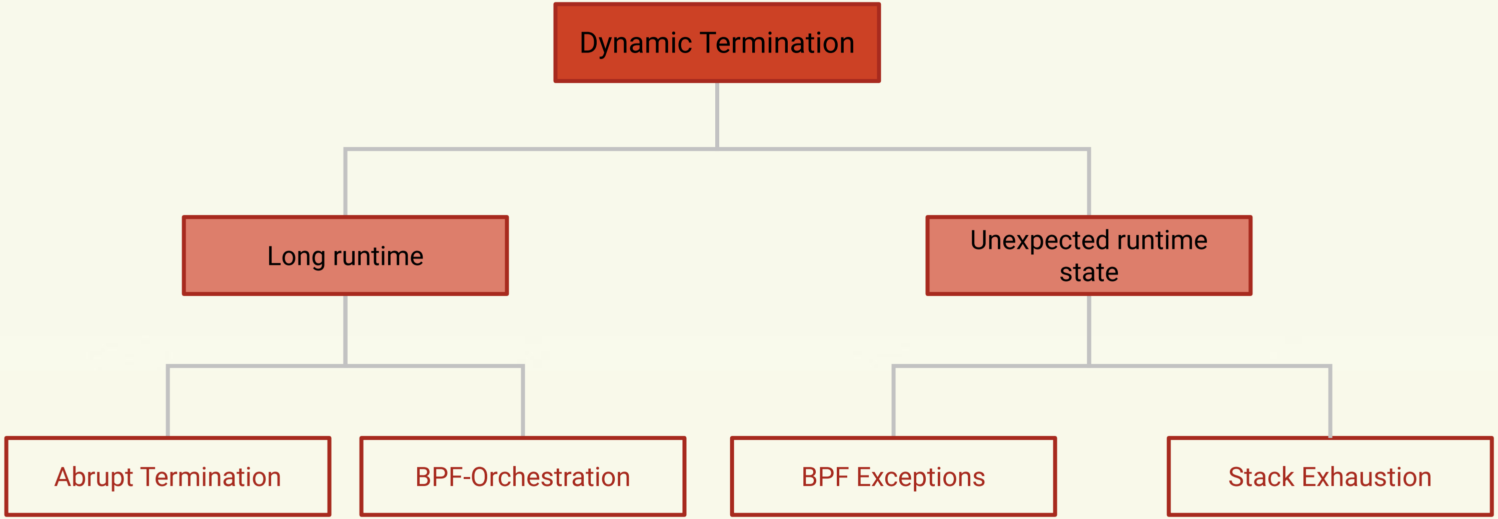
Long runtime

Unexpected runtime state

Abrupt Termination

BPF-Orchestration

When do we need Dynamic Termination



Dynamic Termination

Long runtime

Unexpected runtime state

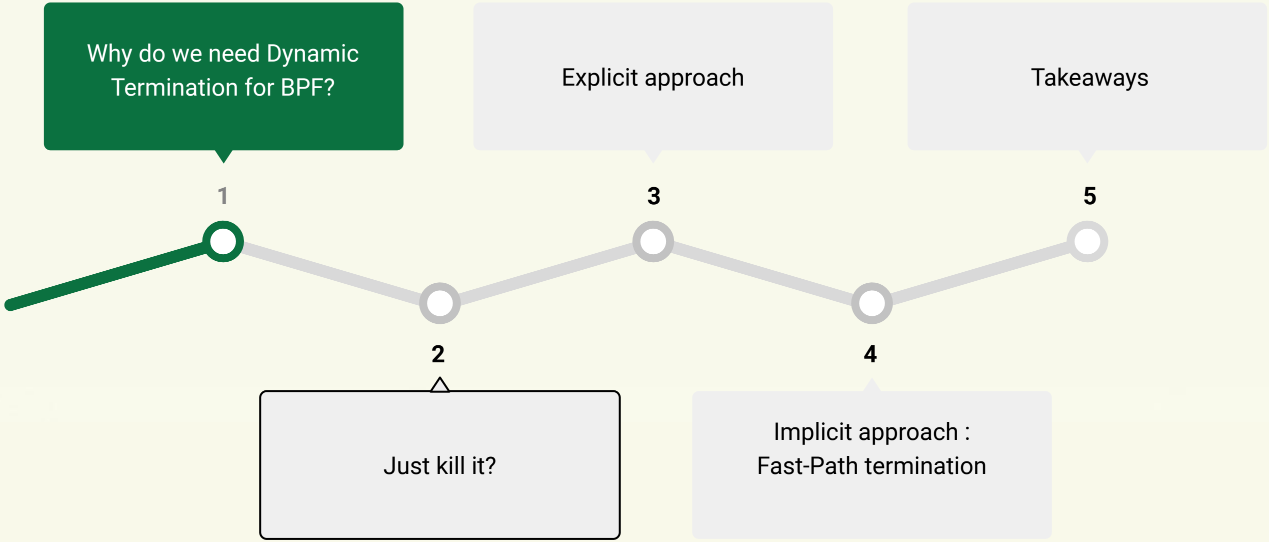
Abrupt Termination

BPF-Orchestration

BPF Exceptions

Stack Exhaustion

ROADMAP



1 Why do we need Dynamic Termination for BPF?

2 Just kill it?

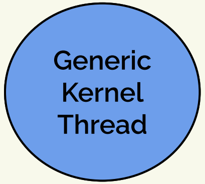
3 Explicit approach

4 Implicit approach : Fast-Path termination

5 Takeaways

Just kill it?

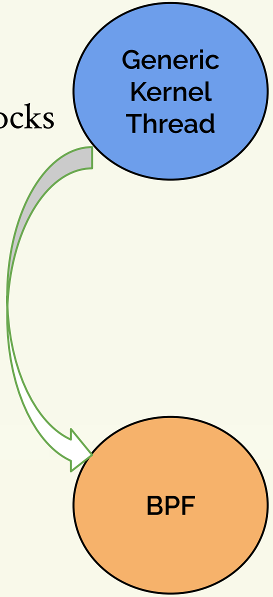
Aborting can lead to memory leaks/deadlocks



Generic Kernel Thread

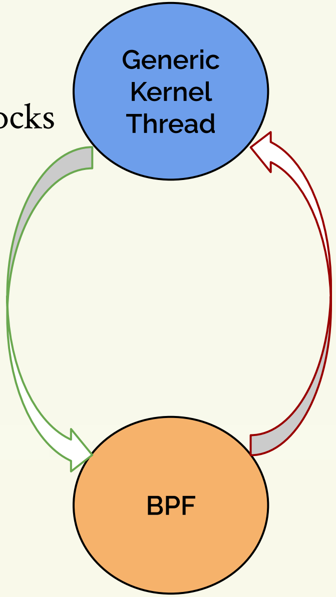
Just kill it?

Aborting can lead to memory leaks/deadlocks



Verifier bookkeeping : live resources known at any point

Just kill it?



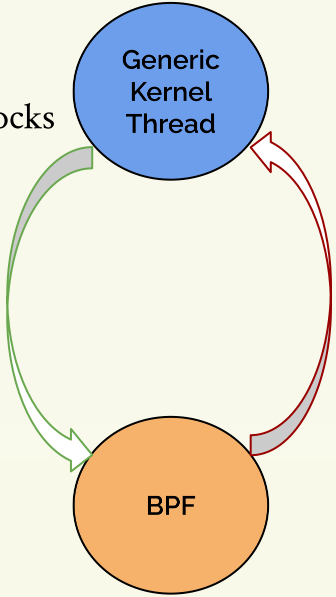
Aborting can lead to memory leaks/deadlocks

Verifier bookkeeping : live resources known at any point

Helper calls takes back to uncharted territory

Just kill it?

Aborting can lead to memory leaks/deadlocks



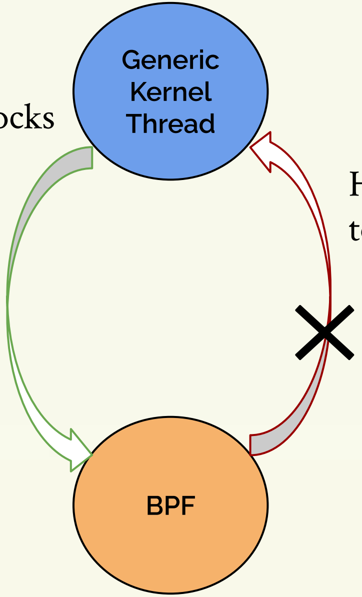
Helper calls takes back to uncharted territory

Verifier bookkeeping : live resources known at any point

Q1. How to track lifetimes for cleanup during termination(unwinding)?

Just kill it?

Aborting can lead to memory leaks/deadlocks



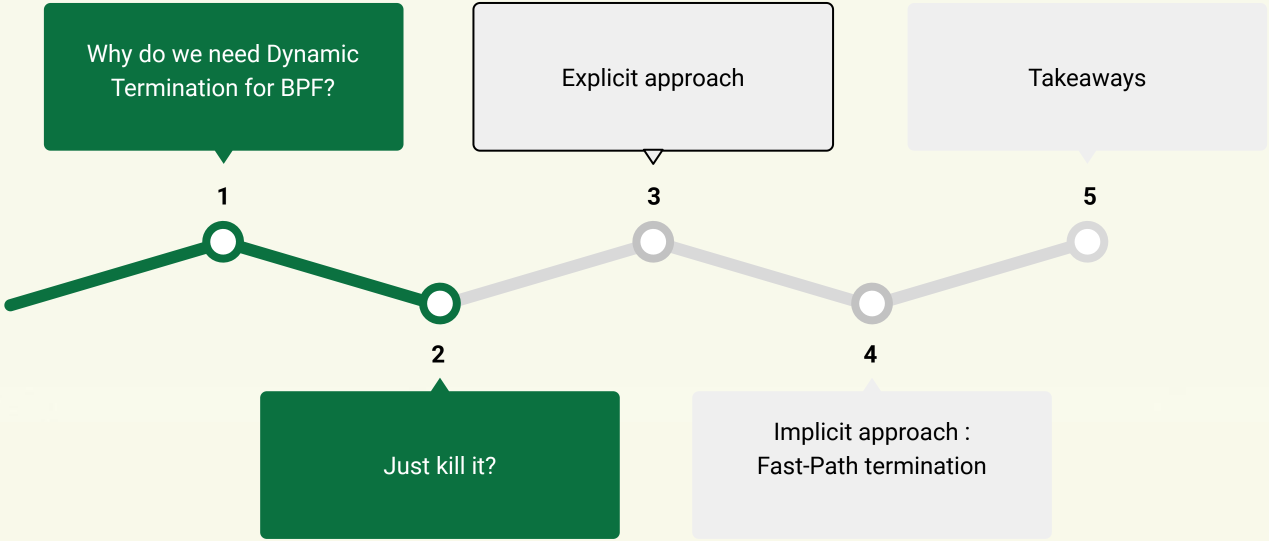
Helper calls takes back to uncharted territory

Verifier bookkeeping : live resources known at any point

Q2. When to trigger unwind ?

Q1. How to track lifetimes for cleanup during termination(unwinding)?

ROADMAP


Why do we need Dynamic Termination for BPF?
1
Just kill it?
2
Explicit approach
3
Implicit approach : Fast-Path termination
4
Takeaways
5

Explicit lifetime-tracking : Garbage collection

- Maintain a list of live objects - alloc list
- For termination, iterate and free when **safe**

– Incurs costs even for no-termination

– Does not utilize verifier's bookkeeping information !

Taking advantage of verifier : Unwind Table

- C++ style unwinding : pre-generate landing pads.
- Industry standard for dealing with cleanups

+ Zero cost for no-termination.

Triggering Unwind : Safe termination points

- For explicit lifetime management, cannot terminate when inside a helper call (helper resources are untracked)
- Any point in BPF text is safe
- Approaches :
    1. Flag check : Runtime Overhead
    2. Kprobes : Zero-cost for no-termination

Termination Approach	Tracking Lifetime	Triggering Unwind
Explicit	GC/Unwind Table	Safe Termination Points

Table : Dynamic Termination

Shortcomings of Explicit resource management

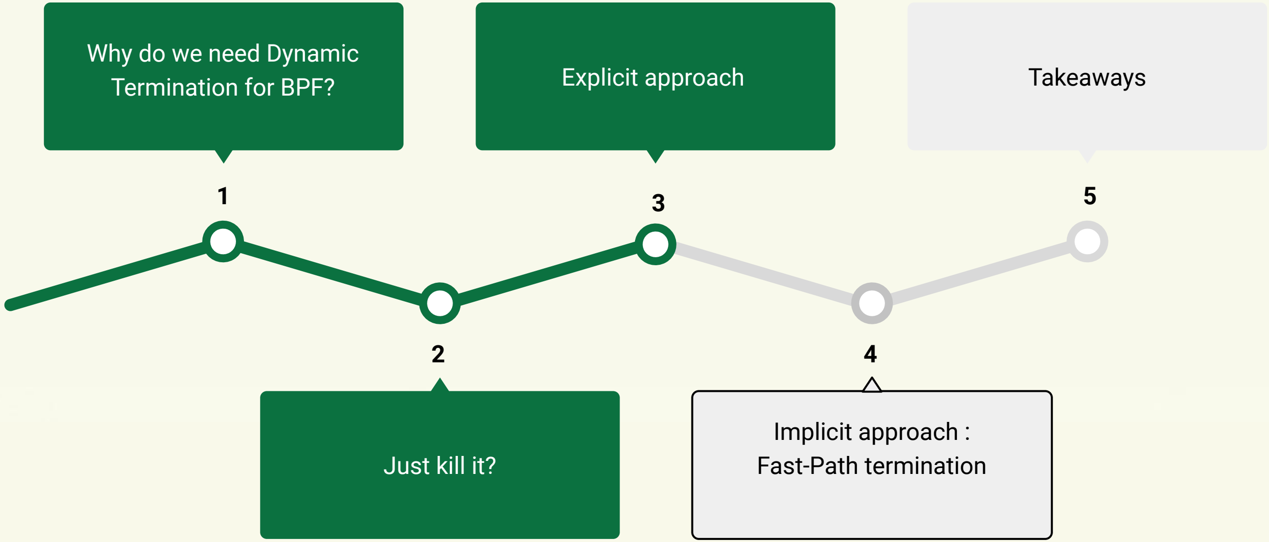
Garbage Collection

- Runtime overhead for no-termination

Unwind Table

- Complexity : Sync unwind table with BPF→x86 translation.
  - Inlining
  - Dead-Code elimination
  - JIT optimizations
- Correctness problem unless table verified.
- Weakens memory safety guarantee.

ROADMAP



1 Why do we need Dynamic Termination for BPF?

2 Just kill it?

3 Explicit approach

4 Implicit approach : Fast-Path termination

5 Takeaways

Revisiting the BPF advantage

1. C has no lifetime management. ⇐ Garbage Collection approach

Revisiting the BPF advantage

1. C has no lifetime management. ⇐ Garbage Collection approach

2. BPF verifier introduces/manages lifetime of objects. ⇐ Landing Pad approach

Revisiting the BPF advantage

1. C has no lifetime management. ⇐ Garbage Collection approach

2. BPF verifier introduces/manages lifetime of objects. ⇐ Landing Pad approach

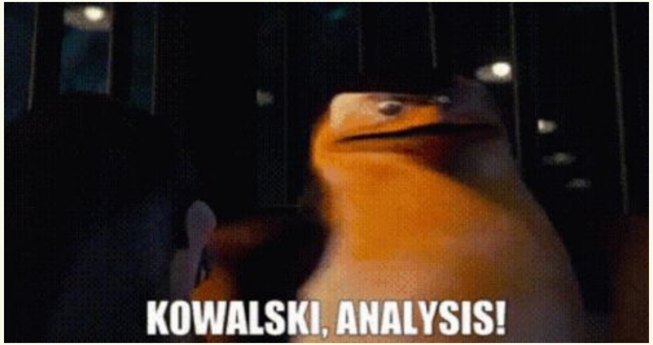
3. Additionally, the verifier also restricts control flow ⇐ **Can we leverage this?**
   ↳ No infinite loops through back-edges

Revisiting the BPF advantage

1. C has no lifetime management. ⇐ Garbage Collection approach

2. BPF verifier introduces/manages lifetime of objects. ⇐ Landing Pad approach

3. Additionally, the verifier also restricts control flow ⇐ **Can we leverage this?**
   ㄴ No infinite loops through back-edges



Implicit Lifetime Management

- Verified BPF program’s control flow encodes cleanup

- $\underbrace{\text{Accelerated execution}}_{\textbf{Fast-Path}}$ to terminate after releasing any live resources

Fast-Path Termination

Dynamically patching target BPF program with a faster version.

Fast-Path Termination

Dynamically patching target BPF program with a faster version.

- Patch all helper calls to create a fall-through.

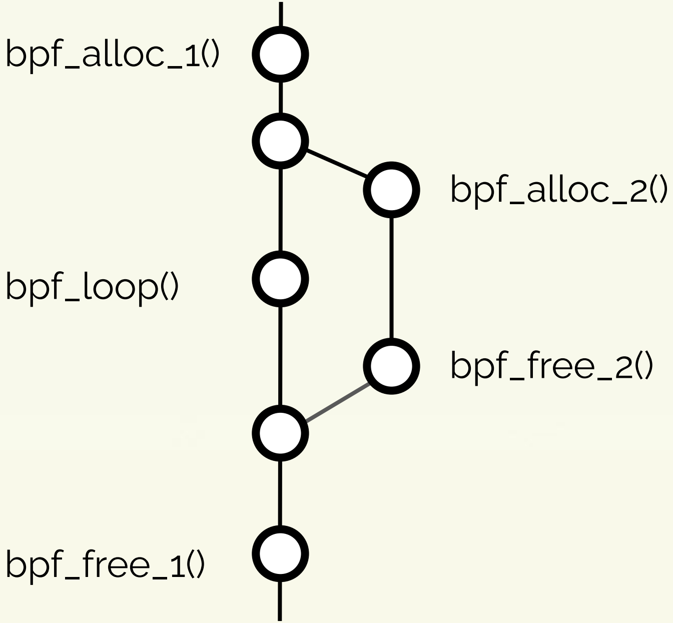
- Keep helpers which free resources to release objects allocated before termination request.

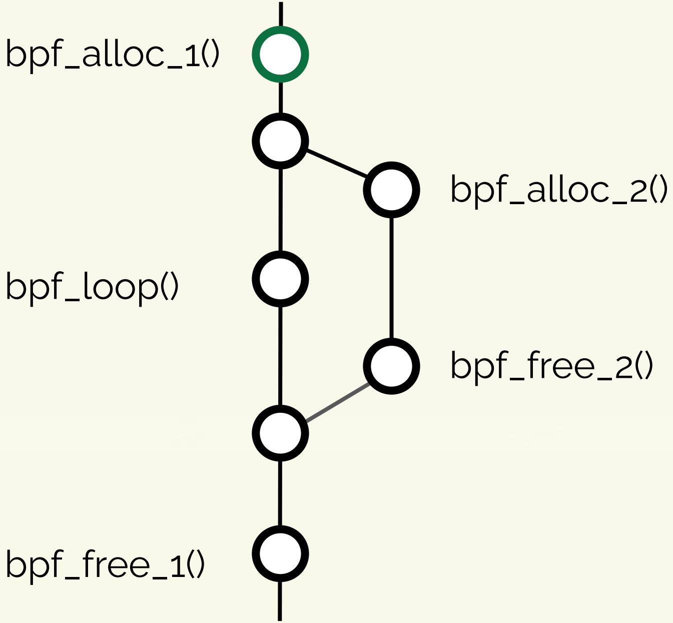
Fast-Path Termination

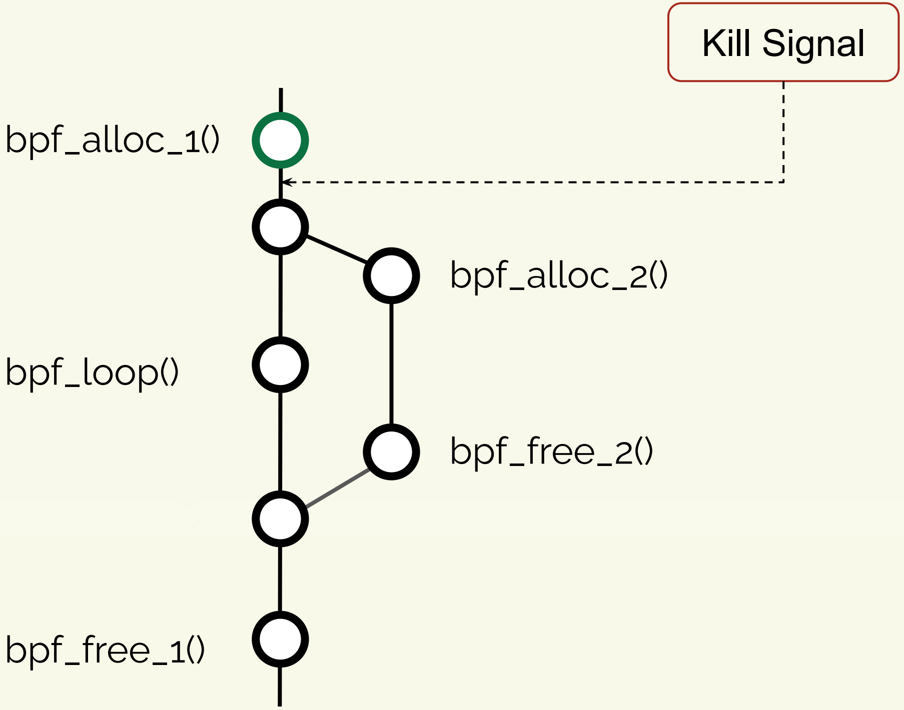
Dynamically patching target BPF program with a faster version.

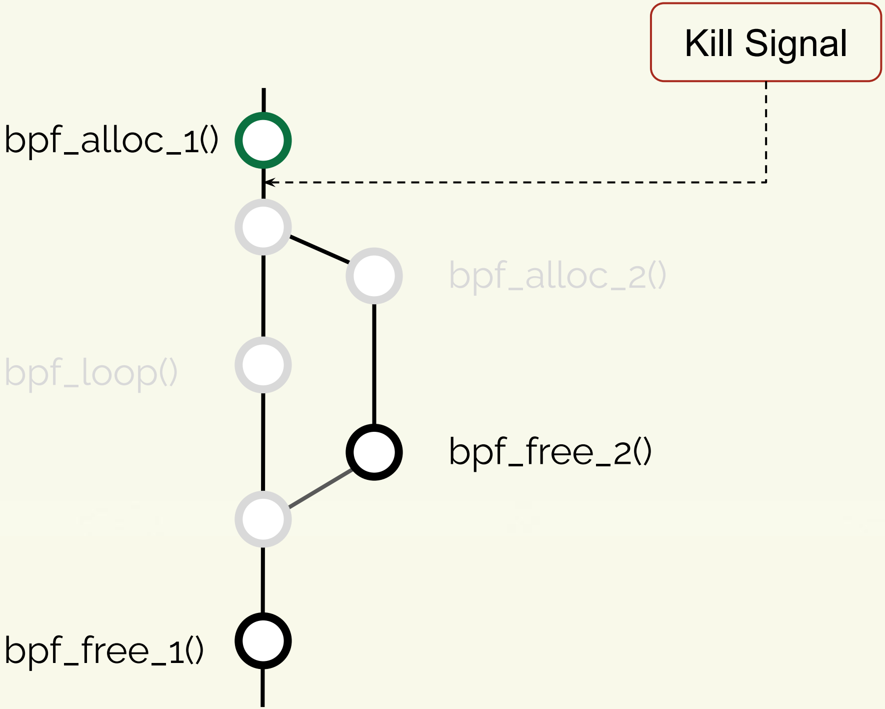
- Patch all helper calls to create a fall-through. *(Leverage verifier's control flow restrictions)*

- Keep helpers which free resources to release objects allocated before termination request. *(Leverage verifier's lifecycle management)*

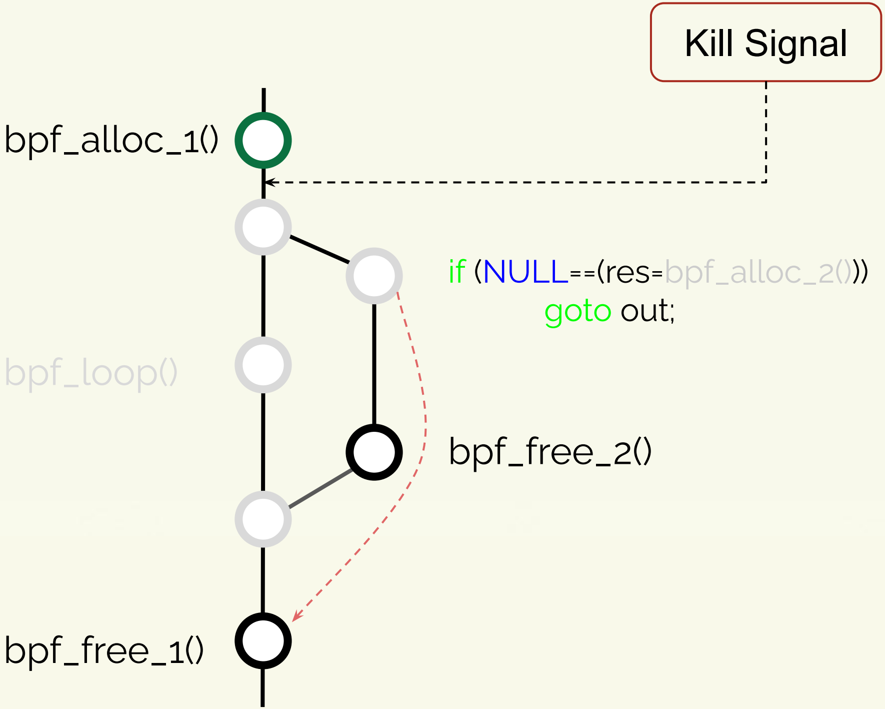

bpf_alloc_1()
bpf_alloc_2()
bpf_loop()
bpf_free_2()
bpf_free_1()


bpf_alloc_1()
bpf_alloc_2()
bpf_loop()
bpf_free_2()
bpf_free_1()

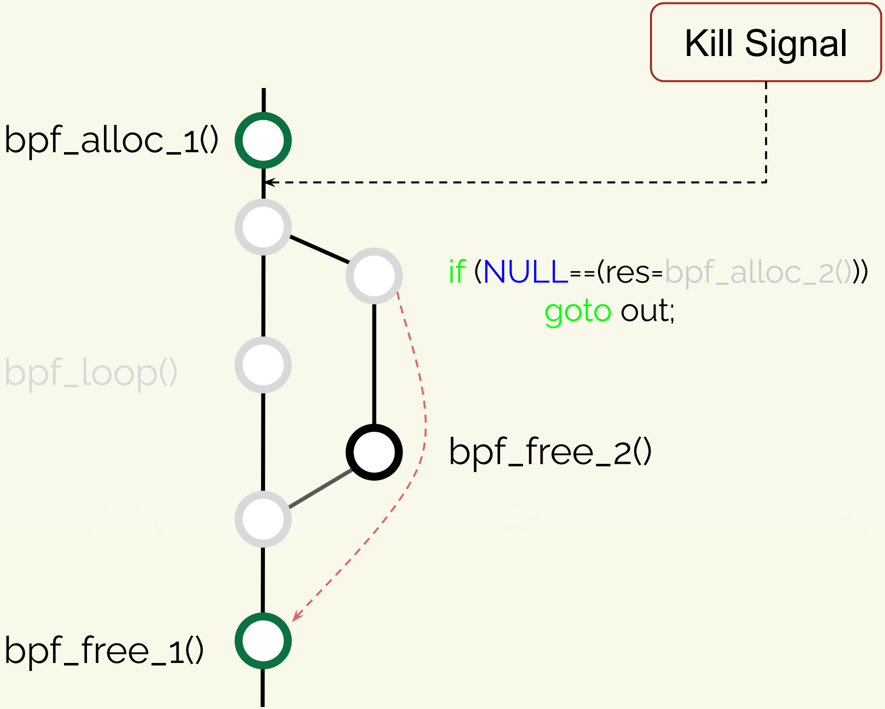

Kill Signal
bpf_alloc_1()
bpf_alloc_2()
bpf_loop()
bpf_free_2()
bpf_free_1()



- Stripped BPF program only has simple BPF insns to execute

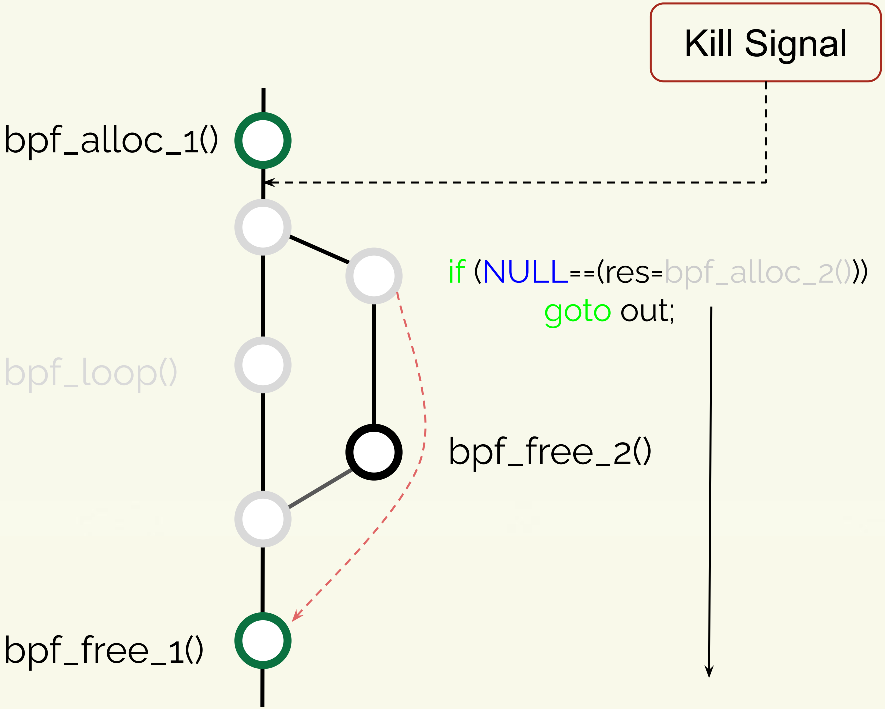


- Stripped BPF program only has simple BPF insns to execute
- Patched program takes nearest exit routes => *Fast fallthrough*



```
if (NULL==(res=bpf_alloc_2()))
        goto out;
```

- Stripped BPF program only has simple BPF insns to execute
- Patched program takes nearest exit routes => *Fast fallthrough*
- Pre-termination objects will always be released <= *Implicit Lifetime Management*



- Stripped BPF program only has simple BPF insns to execute
- Patched program takes nearest exit routes => *Fast fallthrough*
- Pre-termination objects will always be released <= *Implicit Lifetime Management*

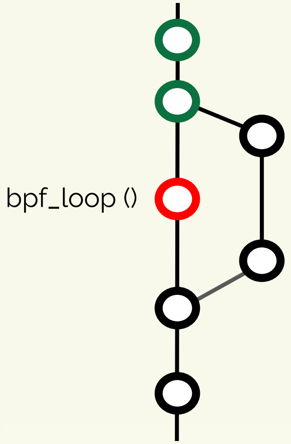
Assumption : Helpers returning a resource always has a failure case checked by the programmer.

Triggering Unwind : Atomic Program Patch

- Patching at runtime demands instruction-level atomicity.
- Halt execution → Apply patch → Resume
- Approaches : Mechanisms used for Safe-Termination Points (flag, kprobes)

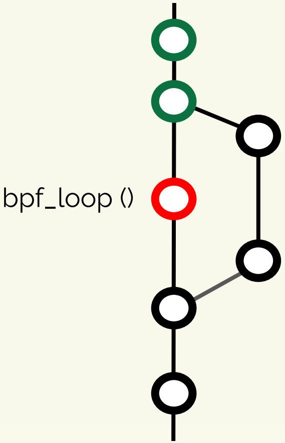
Fall-through for long running helpers

- bpf_for_each_map_elem
  - Iterates through each element in map and calls a callback function
- bpf_loop
  - Bounded loop on a callback function
- bpf_user_ringbuf_drain
  - Invoke a callback for each sample in a user ring buffer.
- bpf_find_vma
  - maps an address of a task to the vma (vm_area_struct) for this address, and feed the vma to a callback BPF function.



Fall-through for long running helpers

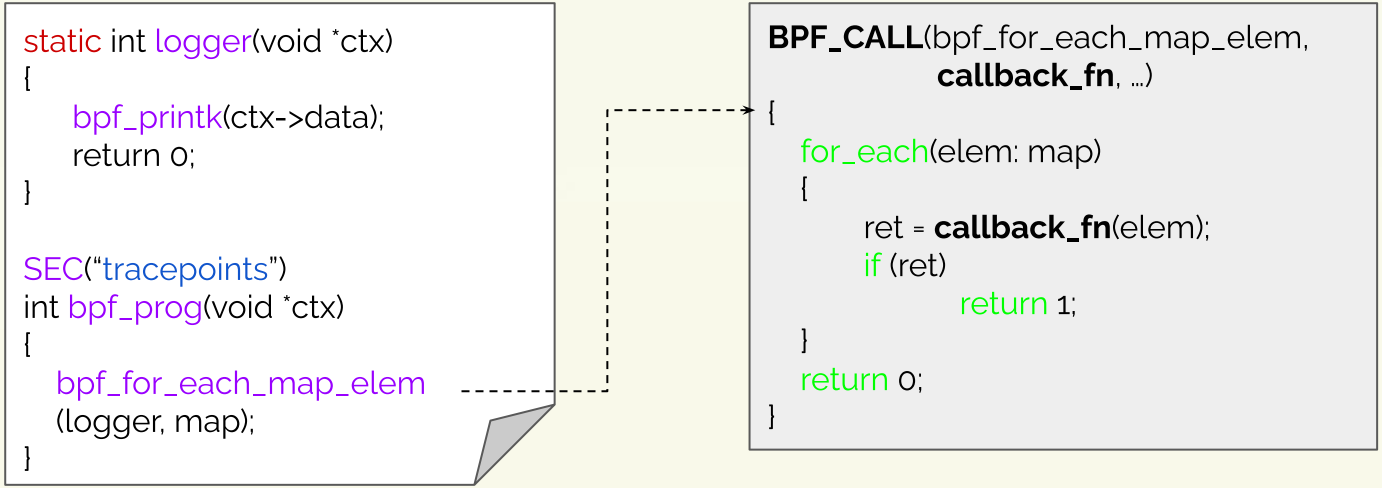
- bpf_for_each_map_elem
  - Iterates through each element in map and calls a callback function
- bpf_loop
  - Bounded loop on a callback function
- bpf_user_ringbuf_drain
  - Invoke a callback for each sample in a user ring buffer.
- bpf_find_vma
  - maps an address of a task to the vma (vm_area_struct) for this address, and feed the vma to a callback BPF function.



Fast-Path for long running helpers

bpf_for_each_map_elem
bpf_loop
bpf_user_ringbuf_drain

BPF program decides whether to continue execution



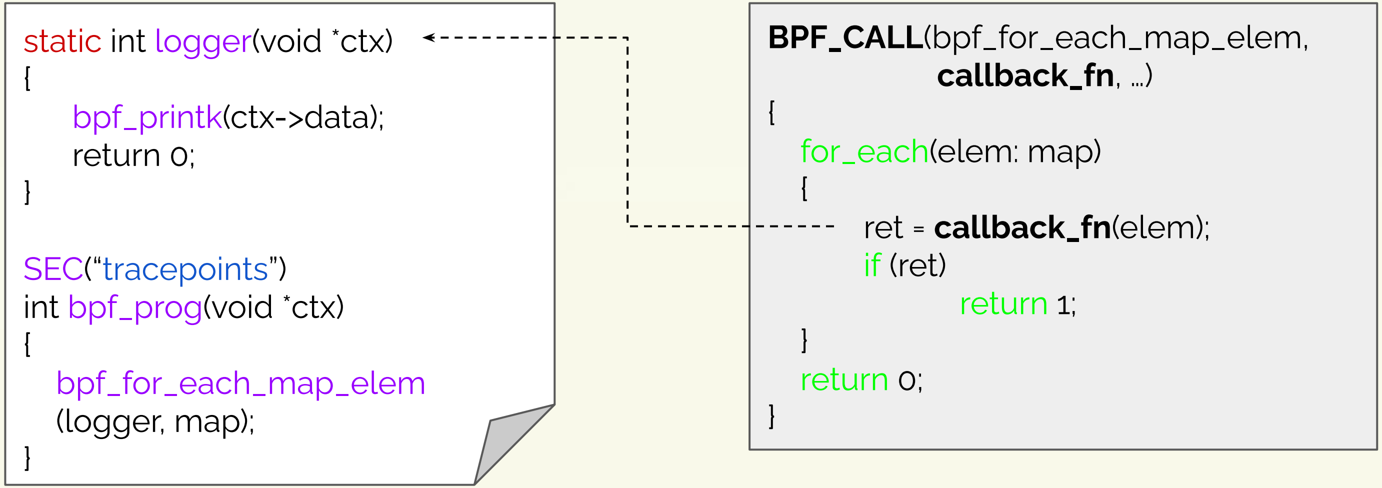
```
static int logger(void *ctx)
{
      bpf_printk(ctx->data);
      return 0;
}

SEC("tracepoints")
int bpf_prog(void *ctx)
{
    bpf_for_each_map_elem
    (logger, map);
}
```

```
BPF_CALL(bpf_for_each_map_elem,
                    callback_fn, ...)
{
   for_each(elem: map)
   {
          ret = callback_fn(elem);
          if (ret)
                 return 1;
   }
   return 0;
}
```

Fast-Path for long running helpers

bpf_for_each_map_elem
bpf_loop
bpf_user_ringbuf_drain

} **BPF program decides whether to continue execution**



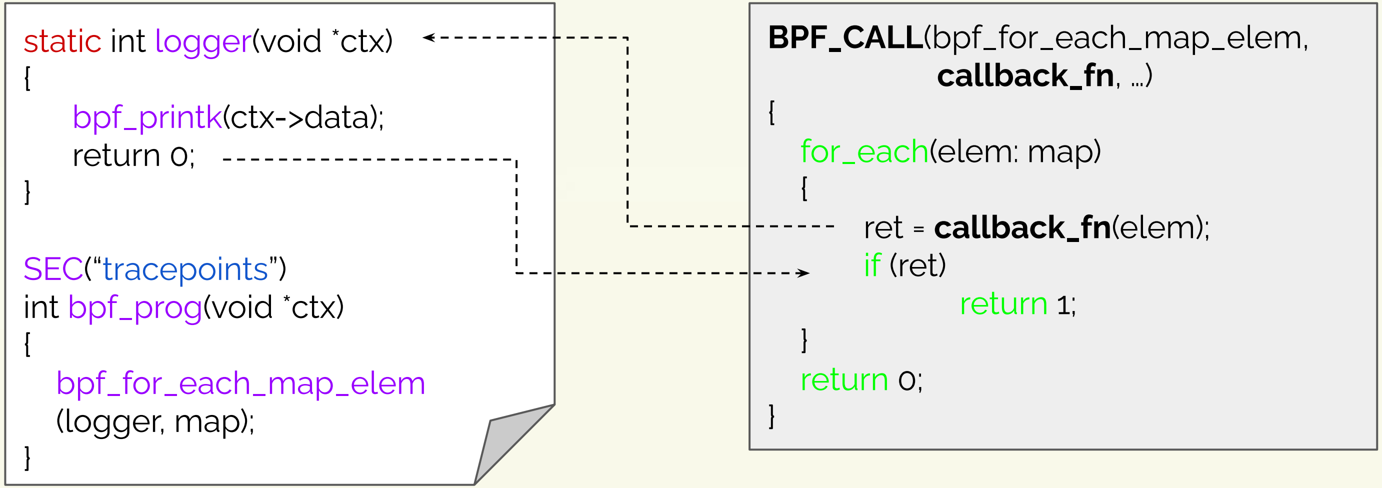
```
static int logger(void *ctx)
{
     bpf_printk(ctx->data);
     return 0;
}

SEC("tracepoints")
int bpf_prog(void *ctx)
{
   bpf_for_each_map_elem
   (logger, map);
}
```

```
BPF_CALL(bpf_for_each_map_elem,
                  callback_fn, ...)
{
   for_each(elem: map)
   {
         ret = callback_fn(elem);
         if (ret)
                  return 1;
   }
   return 0;
}
```

Fast-Path for long running helpers

bpf_for_each_map_elem
bpf_loop
bpf_user_ringbuf_drain

BPF program decides whether to continue execution



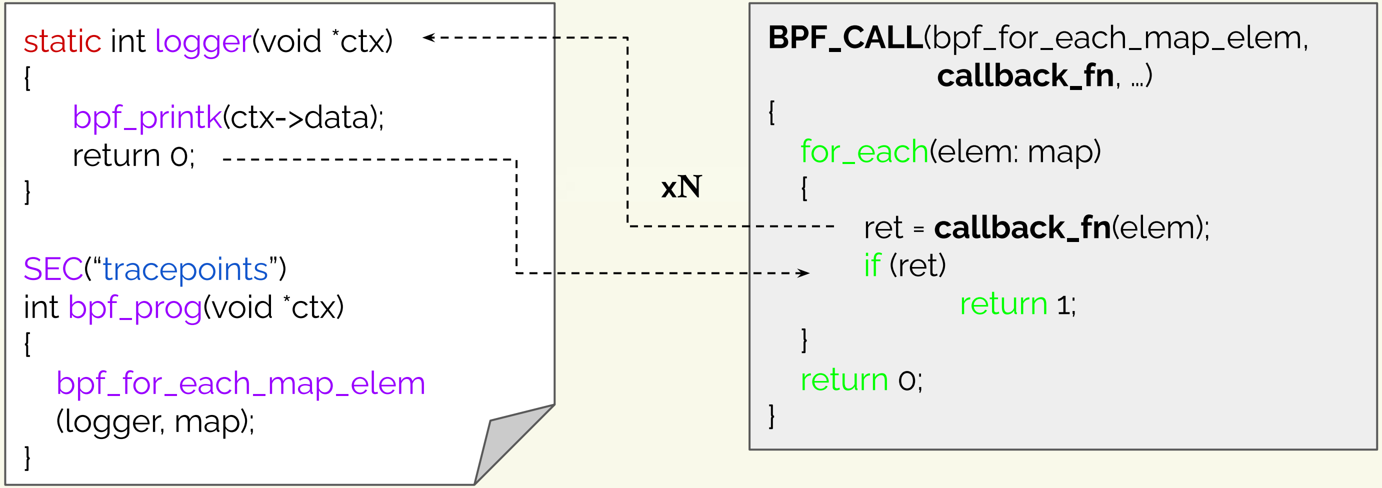
```
static int logger(void *ctx)
{
     bpf_printk(ctx->data);
     return 0;
}

SEC("tracepoints")
int bpf_prog(void *ctx)
{
    bpf_for_each_map_elem
    (logger, map);
}
```

```
BPF_CALL(bpf_for_each_map_elem,
                   callback_fn, ...)
{
   for_each(elem: map)
   {
         ret = callback_fn(elem);
         if (ret)
                   return 1;
   }
   return 0;
}
```

Fast-Path for long running helpers

bpf_for_each_map_elem
bpf_loop
bpf_user_ringbuf_drain

BPF program decides whether to continue execution



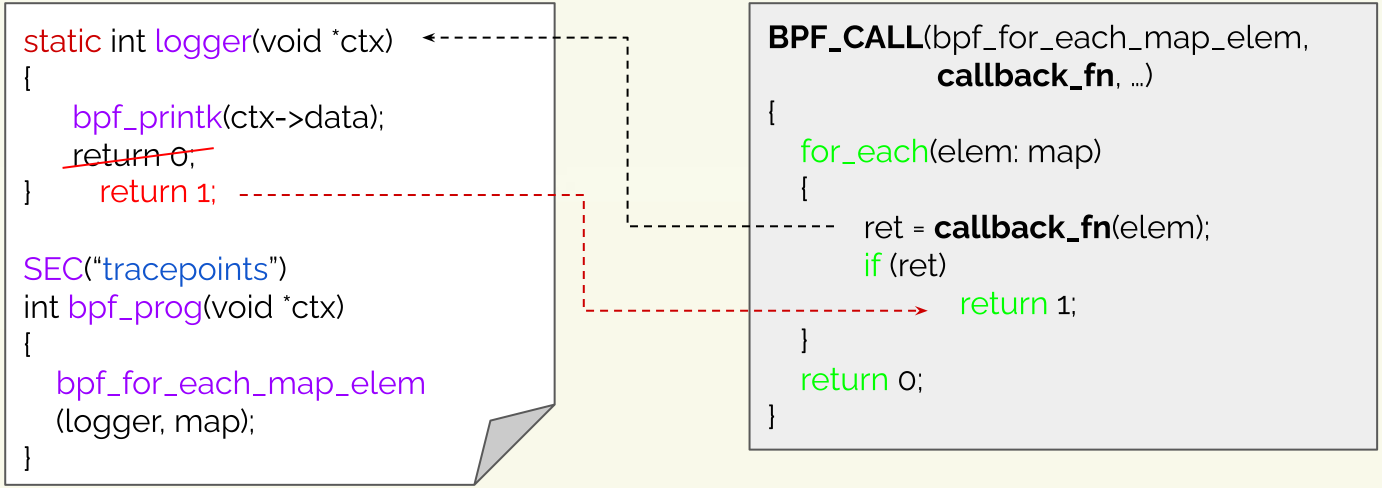
```
static int logger(void *ctx)
{
     bpf_printk(ctx->data);
     return 0;
}

SEC(“tracepoints”)
int bpf_prog(void *ctx)
{
   bpf_for_each_map_elem
   (logger, map);
}
```

```
BPF_CALL(bpf_for_each_map_elem,
                 callback_fn, …)
{
   for_each(elem: map)
   {
         ret = callback_fn(elem);
         if (ret)
                   return 1;
   }
   return 0;
}
```

Fast-Path for long running helpers

bpf_for_each_map_elem
bpf_loop
bpf_user_ringbuf_drain

BPF program decides whether to continue execution

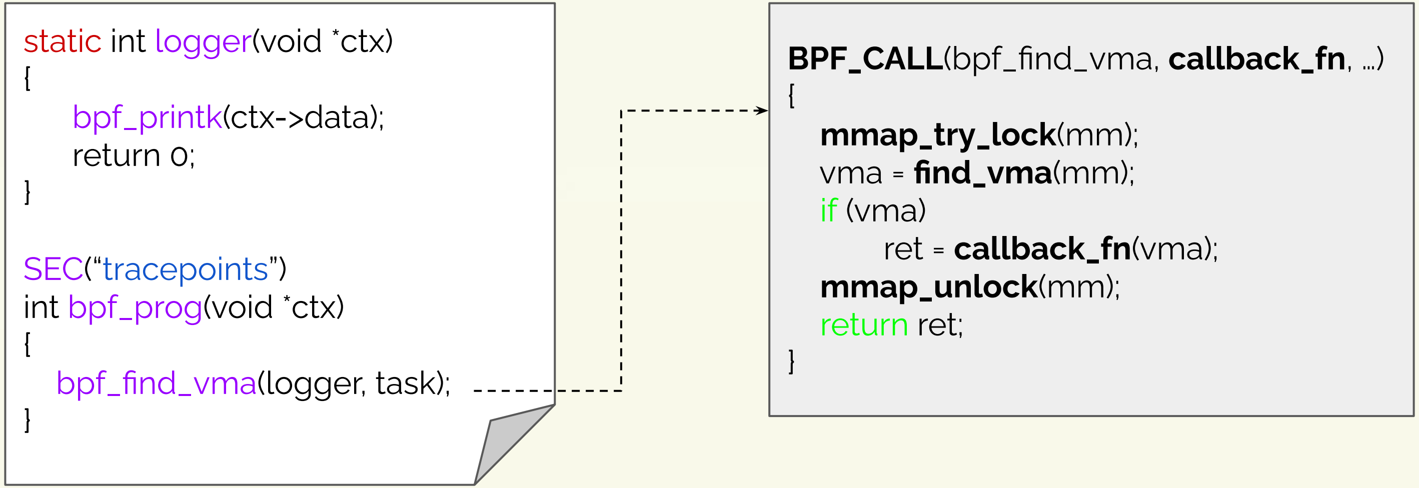


```
static int logger(void *ctx)
{
     bpf_printk(ctx->data);
     ~~return 0;~~
}       return 1;

SEC("tracepoints")
int bpf_prog(void *ctx)
{
   bpf_for_each_map_elem
   (logger, map);
}
```

```
BPF_CALL(bpf_for_each_map_elem,
                callback_fn, ...)
{
   for_each(elem: map)
   {
         ret = callback_fn(elem);
         if (ret)
               return 1;
   }
   return 0;
}
```

Fast-Path for long running helpers

bpf_find_vma } **Just a long running helper; BPF program cannot request to prematurely exit.**

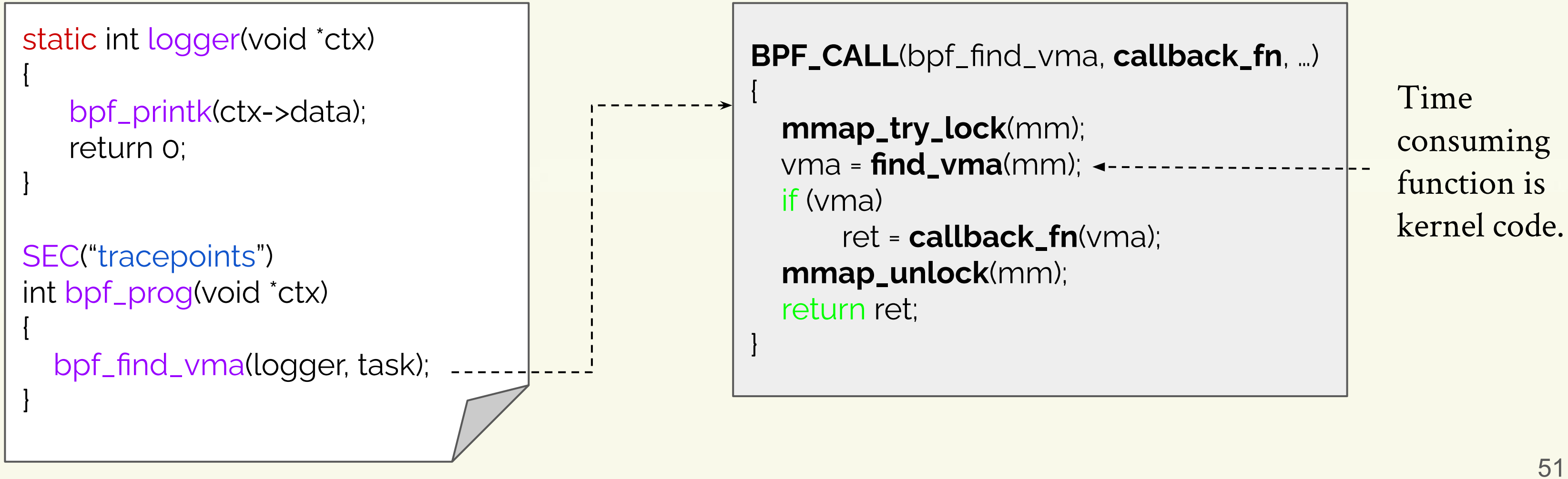


```
static int logger(void *ctx)
{
      bpf_printk(ctx->data);
      return 0;
}

SEC("tracepoints")
int bpf_prog(void *ctx)
{
    bpf_find_vma(logger, task);
}
```

```
BPF_CALL(bpf_find_vma, callback_fn, ...)
{
    mmap_try_lock(mm);
    vma = find_vma(mm);
    if (vma)
            ret = callback_fn(vma);
    mmap_unlock(mm);
    return ret;
}
```

Fast-Path for long running helpers

bpf_find_vma } **Just a long running helper; BPF program cannot request to prematurely exit.**



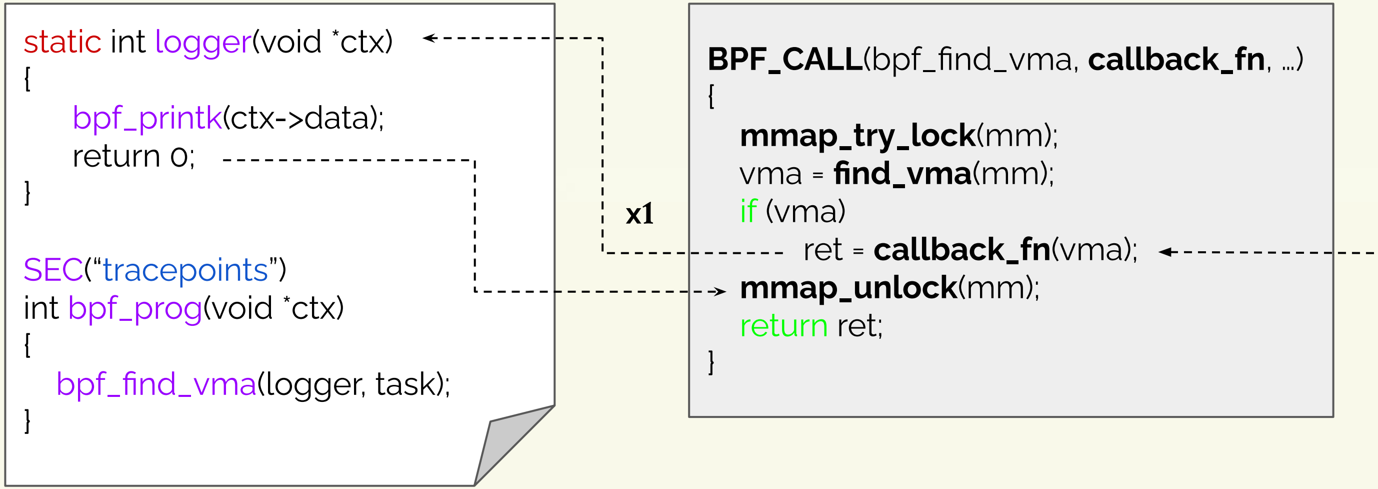
```
static int logger(void *ctx)
{
     bpf_printk(ctx->data);
     return 0;
}

SEC("tracepoints")
int bpf_prog(void *ctx)
{
    bpf_find_vma(logger, task);
}
```

```
BPF_CALL(bpf_find_vma, callback_fn, ...)
{
   mmap_try_lock(mm);
   vma = find_vma(mm);
   if (vma)
          ret = callback_fn(vma);
   mmap_unlock(mm);
   return ret;
}
```

Time consuming function is kernel code.

Fast-Path for long running helpers

bpf_find_vma } Just a long running helper; BPF program cannot request to prematurely exit.



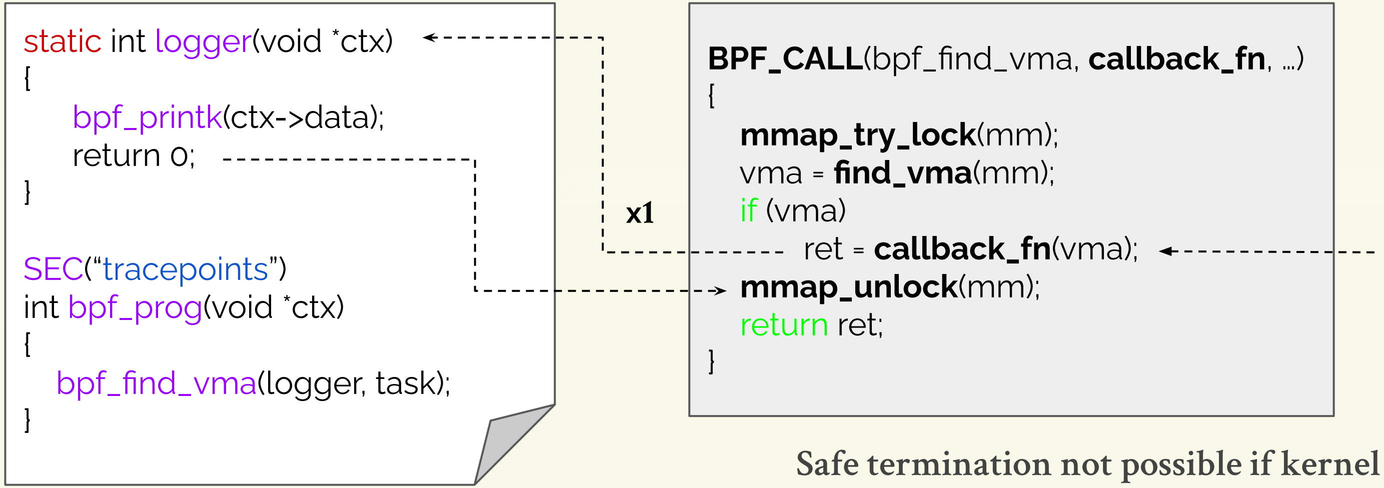
```
static int logger(void *ctx)
{
        bpf_printk(ctx->data);
        return 0;
}

SEC("tracepoints")
int bpf_prog(void *ctx)
{
    bpf_find_vma(logger, task);
}
```

```
BPF_CALL(bpf_find_vma, callback_fn, ...)
{
    mmap_try_lock(mm);
    vma = find_vma(mm);
    if (vma)
        ret = callback_fn(vma);
    mmap_unlock(mm);
    return ret;
}
```

Callback_fn is only called once at the end.

Fast-Path for long running helpers

bpf_find_vma } **Just a long running helper; BPF program cannot request to prematurely exit.**



```
static int logger(void *ctx)
{
     bpf_printk(ctx->data);
     return 0;
}

SEC("tracepoints")
int bpf_prog(void *ctx)
{
   bpf_find_vma(logger, task);
}
```

```
BPF_CALL(bpf_find_vma, callback_fn, ...)
{
   mmap_try_lock(mm);
   vma = find_vma(mm);
   if (vma)
        ret = callback_fn(vma);
   mmap_unlock(mm);
   return ret;
}
```

Callback_fn is only called once at the end.

Safe termination not possible if kernel code is cause of delay!

Making BPF termination compliant

Critical helpers/kfuncs must have error codes which a programmer has to check before proceeding.

{bpf_spin_lock, bpf_refcount_acquire} currently does not comply !

```
SEC("tc")
int bpf_prog(void *ctx)
{
    // obtain lock
    bpf_spin_lock(lock);
    // Critical Section
    bpf_spin_unlock(lock);
}
```

Verifier assumes a spin_lock will always succeed.

```
SEC("tc")
int bpf_prog(void *ctx)
{
    // obtain lock
    ret = bpf_spin_lock(lock);
    if (!ret)
    {
        // Critical Section
        bpf_spin_unlock(lock);
    }
}
```

*Proposed change will ensure a program does not enter CS when **lock** returns prematurely on termination*

Advantages :

- No need to have a new program (landing pads) for cleanups.
  - Allocated resources will auto-cleanup from unpatched free-up helper calls.

- Complexity of managing resources as per JIT/Verifier optimization of BPF insns is removed.

- Memory safety property cannot be compromised.

Advantages :

- No need to have a new program (landing pads) for cleanups.
  - Allocated resources will auto-cleanup from unpatched free-up helper calls.

- Complexity of managing resources as per JIT/Verifier optimization of BPF insns is removed.

- Memory safety property cannot be compromised.

Limitations

- The error check from API changes puts more burden on a BPF programmer.

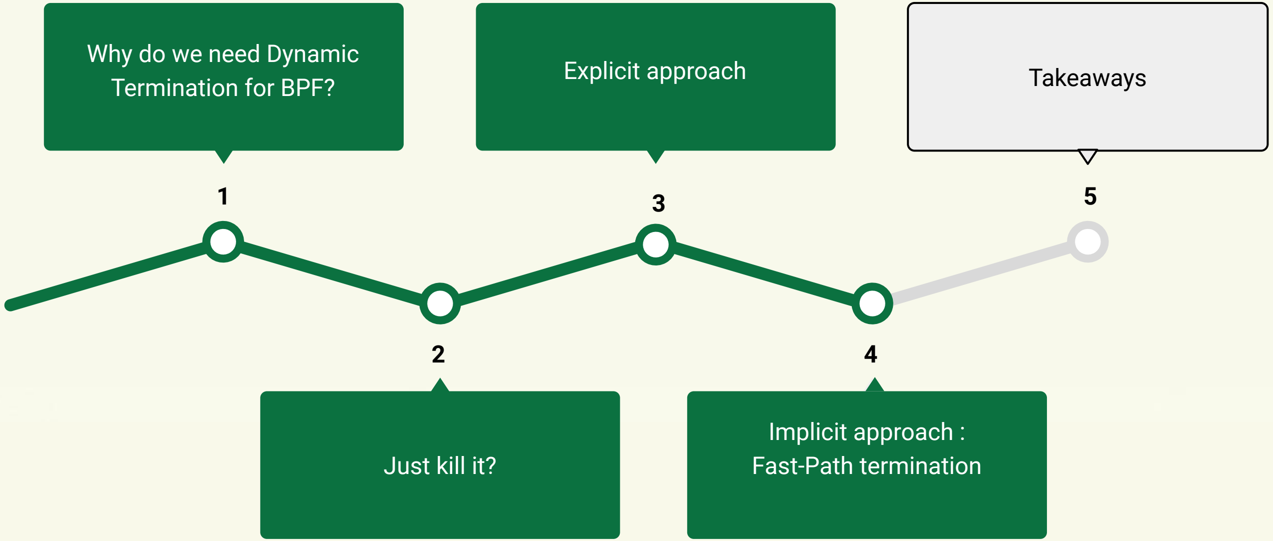
- Termination is not immediate as non-helpers are still executed.

- Kptrs, acquired before termination, can still get modified.
  However, programmed checks can safeguard against termination-time unexpected modifications.

Termination Approach	Tracking Lifetime	Triggering Unwind
Explicit	GC/Unwind Table	Safe Termination Points
Implicit	Fast-Path	Atomic Program Patch

Table : Dynamic Termination

ROADMAP



1 Why do we need Dynamic Termination for BPF?

2 Just kill it?

3 Explicit approach

4 Implicit approach : Fast-Path termination

5 Takeaways

Takeaways : Fast-Path Termination

1. Leverages encoded cleanup & control-flow restrictions.

2. Patch BPF program to accelerate execution.

3. Long running helpers switching between BPF-kernel support early exit through return values.

Summary

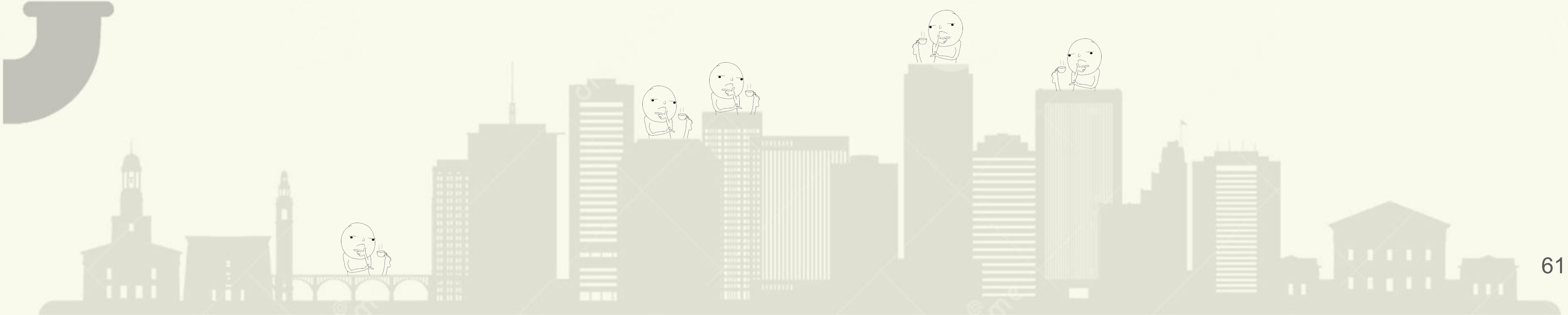
1. BPF termination is a two-part problem :
   1. How to track live objects ?
   2. How/When to trigger unwind ?

2. Explicit management had its shortcomings : complexity, overheads, etc.

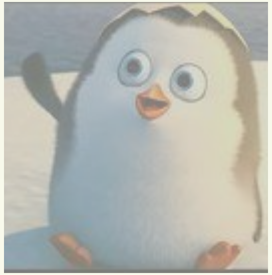
3. Proposed Fast-Path termination.

4. Call for making all helpers/kfuncs termination complaint.

Questions ?



Thank You



Backup Slides

Dealing with Loop inlining

Based on certain conditions (non-constant callback_fn, non-zero flag, etc) a bpf_loop can be inlined.

Dealing with Loop inlining

Based on certain conditions (non-constant callback_fn, non-zero flag, etc) a bpf_loop can be inlined.

```
bpf_loop(10, foo, NULL, 0);        ⇒        for (int i = 0; i < 10; ++i)
                                                foo(i, NULL);
```

Dealing with Loop inlining

Based on certain conditions (non-constant callback_fn, non-zero flag, etc) a bpf_loop can be inlined.

```
for (int i = 0; i < 10; ++i)
    foo(i, NULL);
```

```
/* if reg_loop_cnt >= reg_loop_max skip the loop body */
BPF_JMP_REG(BPF_JGE, reg_loop_cnt, reg_loop_max, 5),

/* callback call*/
BPF_MOV64_REG(BPF_REG_1, reg_loop_cnt),
BPF_MOV64_REG(BPF_REG_2, reg_loop_ctx),
BPF_CALL_REL(0),

/* increment loop counter */
BPF_ALU64_IMM(BPF_ADD, reg_loop_cnt, 1),

/* jump to loop header if callback returned 0 */
BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, -6),
```

Dealing with Loop inlining

Based on certain conditions (non-constant callback_fn, non-zero flag, etc) a bpf_loop can be inlined.

```
~~for (int i = 0; i < 10; ++i)~~
    foo(i, NULL);
```

```
/* if reg_loop_cnt >= reg_loop_max skip the loop body */
BPF_JMP_REG(BPF_JGE, reg_loop_cnt, reg_loop_max, 5),

/* callback call*/
BPF_MOV64_REG(BPF_REG_1, reg_loop_cnt),
BPF_MOV64_REG(BPF_REG_2, reg_loop_ctx),
BPF_CALL_REL(0),

/* increment loop counter */
BPF_ALU64_IMM(BPF_ADD, reg_loop_cnt, 1),

/* jump to loop header if callback returned 0 */
~~BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, -6),~~
               patch with nops to fall-through
```

Insights

- Verifier range analysis ensured any branch decision based on runtime values i.e. helper returns, map values, etc cannot corrupt kernel state or hurt safety.
    - Patching helpers to return error values will still take the program to one of the possible branches which the verifier has already marked as safe to execute.

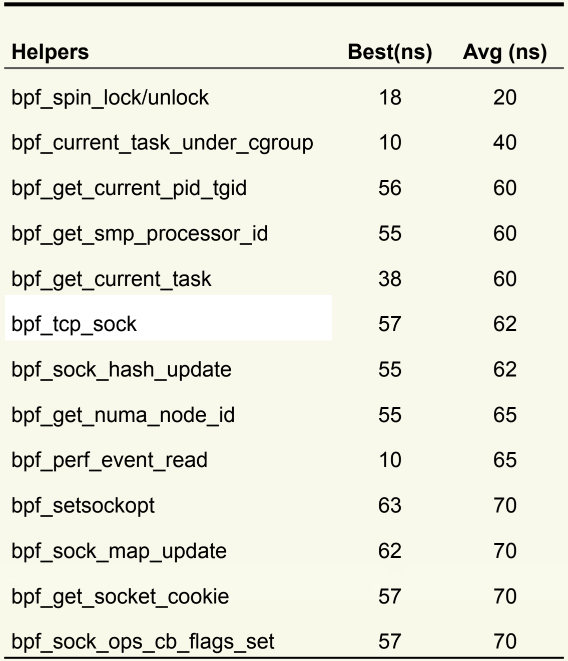
- Stripping-off all helpers will drastically reduce runtime of the BPF program
    - Long running helpers, or helpers in generate cost more than simple BPF insns
    - Currently low Instruction and complexity limit of BPF means an insignificant time to completion for a program with no helpers.

- Modified program will be same structurally. (Replacing helper calls with dummies won't bring any new JIT/Verifier optimization)

- Even if the patched BPF program can write unexpected to a kernel object, the values still would be within an acceptable range from a verified program.
    - Always doing what the verified said is logically okay. Hence the kernel is still safe.

Locating in Design axes :

- Runtime Overhead : **O(Helpers)** ≈ 15 ns * #helpers
- Termination Behaviour
    - Quick/Delayed
    - Memory Requirement
- Programming Cost



Helpers	Best(ns)	Avg (ns)
bpf_spin_lock/unlock	18	20
bpf_current_task_under_cgroup	10	40
bpf_get_current_pid_tgid	56	60
bpf_get_smp_processor_id	55	60
bpf_get_current_task	38	60
bpf_tcp_sock	57	62
bpf_sock_hash_update	55	62
bpf_get_numa_node_id	55	65
bpf_perf_event_read	10	65
bpf_setsockopt	63	70
bpf_sock_map_update	62	70
bpf_get_socket_cookie	57	70
bpf_sock_ops_cb_flags_set	57	70

Raj Sahu and Dan Williams. 2023. Enabling BPF Runtime policies for better BPF management. In Proceedings of the 1st Workshop on eBPF and Kernel Extensions (eBPF '23)

Garbage Collection

Locating in Design axes :

- Runtime Overhead : **O(allocations)** ≈ 30-110 ns * #allocation
  - Memory : **O(allocations)** ≈ 30B * #allocation
- Termination Behaviour
  - Memory Requirement : **None**
- Programming Cost : **Low Complexity, Moderate Code Spread**

Design Goals

CRITICAL

- Safety : Correctly release all acquired resources

IMPORTANT

- Runtime Overhead : Cost paid for no-termination case
- Termination Behaviour : Quick/delayed; Memory Requirement
- Programming Cost : Kernel Complexity, code spread, Baggage on future modifications

Integrating with Use-Cases

1. Abrupt Termination => sys_bpf() or Timers
2. BPF-Orchestration => sys_bpf()
3. BPF Exceptions and Aborts => Called by bpf_throw
4. Stack Exhaustion => Called by kernel

Until Now

Naive Solution

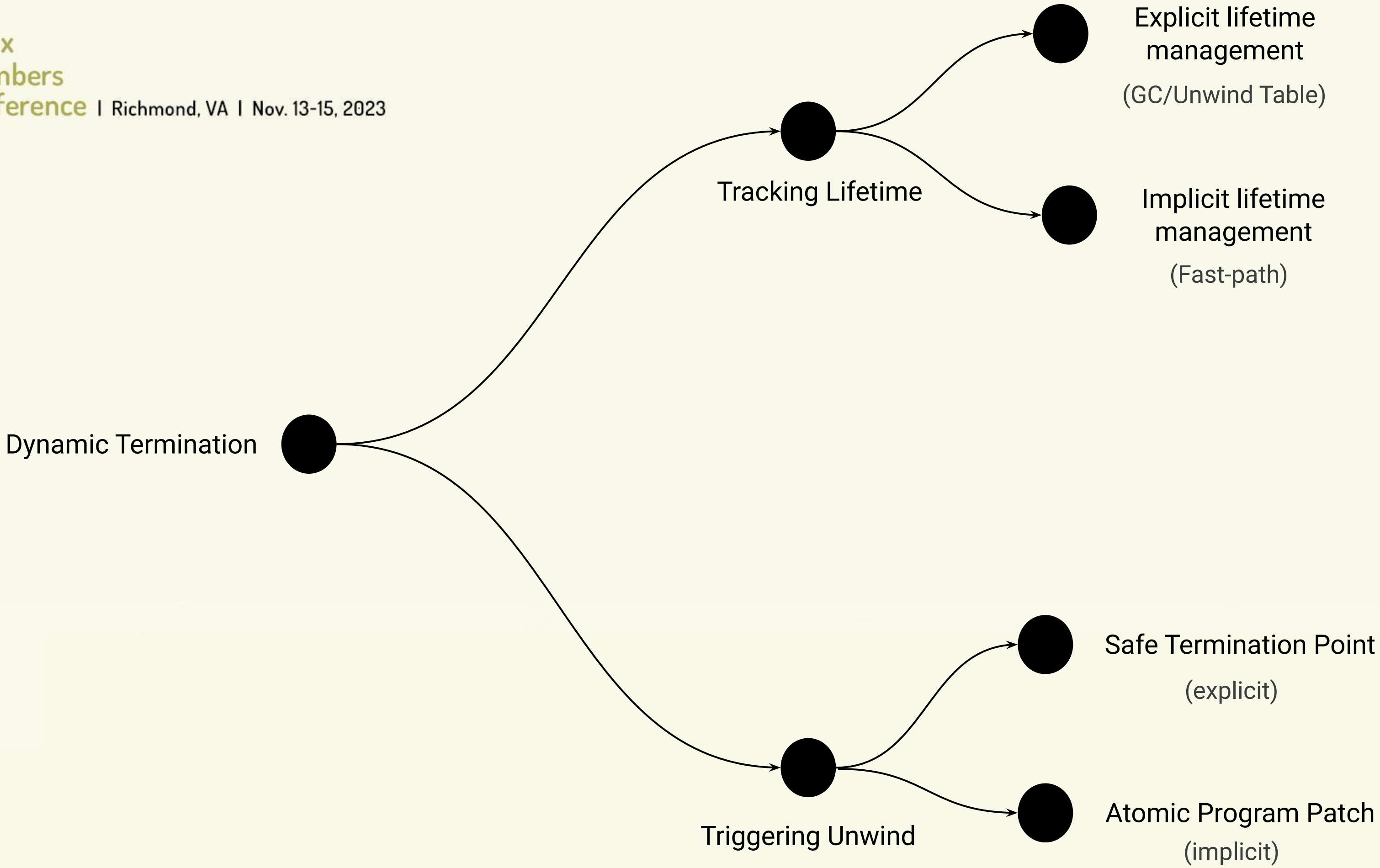
- Runtime Overhead : **HIGH**
- Termination Behaviour :
  - Quick/delayed : **Quick**
  - Memory Requirement : **Zero**
- Programming Cost : **HIGH**

Kprobe Optimization

- Runtime Overhead : **MODERATE**
- Termination Behaviour :
  - Quick/delayed : **Quick**
  - Memory Requirement : **HIGH**
- Programming Cost : **MODERATE**

Cleanup (Unwind Table)

- Runtime Overhead : **Zero**
- Termination Behaviour :
  - Quick/delayed : **Quick**
  - Memory Requirement : **Zero**
- Programming Cost : **HIGH**



Dynamic Termination

Tracking Lifetime

Explicit lifetime management

(GC/Unwind Table)

Implicit lifetime management

(Fast-path)

Triggering Unwind

Safe Termination Point

(explicit)

Atomic Program Patch

(implicit)