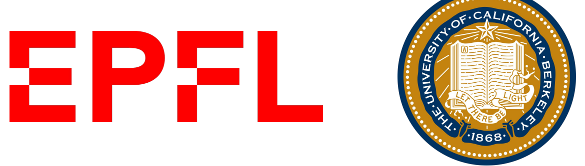
Fast, Flexible, and Practical Kernel Extensions

Kumar Kartikeya Dwivedi, Rishabh Iyer, Sanidhya Kashyap



Introduction

- Kernel extensions allow **safely** customizing an operating system on the fly.
  - Use cases: Observability, Networking, Security, CPU scheduling, etc.
  - Well explored in academia (back in the 90s!), now popular again due to eBPF.

Introduction

- Kernel extensions allow **safely** customizing an operating system on the fly.
  - Use cases: Observability, Networking, Security, CPU scheduling, etc.
  - Well explored in academia (back in the 90s!), now popular again due to eBPF.

- **Kernel interfaces** are how extensions interact with the kernel.
  - Well-defined hooks to handle events in the kernel.
  - Helper functions.
  - Pointers to kernel objects.

Introduction

- Kernel extensions allow **safely** customizing an operating system on the fly.
  - Use cases: Observability, Networking, Security, CPU scheduling, etc.
  - Well explored in academia (back in the 90s!), now popular again due to eBPF.

- **Kernel interfaces** are how extensions interact with the kernel.
  - Well-defined hooks to handle events in the kernel.
  - Helper functions.
  - Pointers to kernel objects.

- Kernel safety is defined w.r.t. kernel interfaces.

Ideal Solution

An ideal kernel extension framework provides **four** properties:

1. **Safety:** Ensuring correct usage of kernel interfaces.

Ideal Solution

An ideal kernel extension framework provides **four** properties:

1. **Safety:** Ensuring correct usage of kernel interfaces.
2. **Flexibility:** Freedom to express diverse functionality.

Ideal Solution

An ideal kernel extension framework provides **four** properties:

1. **Safety:** Ensuring correct usage of kernel interfaces.
2. **Flexibility:** Freedom to express diverse functionality.
3. **Performance:** Low-overhead execution.

Ideal Solution

An ideal kernel extension framework provides **four** properties:

1. **Safety:** Ensuring correct usage of kernel interfaces.
2. **Flexibility:** Freedom to express diverse functionality.
3. **Performance:** Low-overhead execution.
4. **Practicality:** No dependence on a programming language or toolchain.

Problem Statement

- Today, kernel extensibility is either flexible, or performant — not both.
  - We can allow near-arbitrary code, but incur high overhead (e.g. runtime sandboxing).
  - We can severely constrain extension logic, and retain low overhead (e.g. static verification).

Problem Statement

- Today, kernel extensibility is either flexible, or performant — not both.
  - We can allow near-arbitrary code, but incur high overhead (e.g. runtime sandboxing).
  - We can severely constrain extension logic, and retain low overhead (e.g. static verification).

- Existing approaches are inadequate.
  - Trade off flexibility for performance, or vice versa.
  - Bound to a single approach to ensure kernel safety, inheriting its downsides.

SPIN

- Extensions written in Modula-3.
- Kernel safety through type safety.
- Bound to learn and use a single language.
- Accessing data and communication always requires trusted helper calls.
- Expressivity limited to the language.

SPIN - Kernel Interface

```
INTERFACE Strand;

TYPE T <: REFANY;  (* Strand.T is opaque *)

PROCEDURE Block(s:T);
(* Signal to a scheduler that s is not runnable. *)

PROCEDURE Unblock(s: T);
(* Signal to a scheduler that s is runnable. *)

PROCEDURE Checkpoint(s: T);
(* Signal that s is being descheduled and that it
   should save any processor state required for
   subsequent rescheduling. *)

PROCEDURE Resume(s: T);
(* Signal that s is being placed on a processor and
   that it should reestablish any state saved during
   a prior call to Checkpoint. *)

END Strand.
```

VINO

- Relies on runtime sandboxing / SFI.
- All function calls, all memory accesses, checked at runtime.
- Helper functions similar to eBPF, but runtime checked.
- Abort / forcibly terminate when extensions are hung.
- Write mostly arbitrary C++ code.

- Acquired kernel resources push an item in a local buffer.
- On abort, walk through the log and invoke release handlers.
- On normal exit, discard the log.

VINO - Kernel Interface

```
// http server installation,
// invoked at user level
    graftpoint_handle_o *gp;

    gp = graft_namespace->lookup("tcp/80");
    gp->add("http_server.o");
________________________________________

// http server code, run as graft in kernel.
http_server(file_o *fd)
{
    char buf[256];
    fd->read(buf, sizeof(buf));
    // process http request...
    ...
}
```

eBPF

- Extensions submitted as bytecode, language independent (C, Rust).
- Symbolic execution of extension to ensure kernel safety.
- Helper functions, trusted pointers to kernel objects.
- Maps abstracting access to data structures.
- Flexibility constrained by limitations of static verification.
    - E.g. from no loops, to bounded loops, to bpf_loop, to BPF iterators.

- Acquired kernel resources must be released before exit.

eBPF - Kernel Interface

```
s32 BPF_STRUCT_OPS(simple_select_cpu, struct task_struct *p, s32 prev_cpu, u64 wake_flags)
{
        bool is_idle = false;
        s32 cpu;

        cpu = scx_bpf_select_cpu_dfl(p, prev_cpu, wake_flags, &is_idle);
        if (is_idle) {
                stat_inc(0);    /* count local queueing */
                scx_bpf_dispatch(p, SCX_DSQ_LOCAL, SCX_SLICE_DFL, 0);
        }

        return cpu;
}
```

Observation

- Kernel interfaces are **well-defined** and **narrow**.
  - Extensions act on a specific kernel event or object.
  - Well-defined helper functions to access and mutate kernel state.
  - Constrained access to kernel object fields.

Observation

- Kernel interfaces are **well-defined** and **narrow**.
    - Extensions act on a specific kernel event or object.
    - Well-defined helper functions to access and mutate kernel state.
    - Constrained access to kernel object fields.

- Flexibility is mostly about behavior that doesn’t interact with kernel interfaces.
    - Time spent in executing an extension, before termination.
    - How an extension’s *own data* is manipulated.

Observation

- Kernel interfaces are **well-defined** and **narrow**.
  - Extensions act on a specific kernel event or object.
  - Well-defined helper functions to access and mutate kernel state.
  - Constrained access to kernel object fields.
- Flexibility is mostly about behavior that doesn’t interact with kernel interfaces.
  - Time spent in executing an extension, before termination.
  - How an extension’s *own data* is manipulated.
- Static verification amenable to kernel interfaces, but not the rest.

KFlex

- Split kernel safety into two sub-properties:
  - **Kernel Interface Compliance**: Correct usage of kernel interfaces.
  - **Extension Correctness**: Safe usage of extension's own memory, and termination.

- Use **separate** approaches to enforce safety:
  - Static verification for kernel interfaces (i.e. retain eBPF's behavior).
  - Runtime sandboxing + cancellations for extension memory and termination.

- Resources protected by each sub-property have separate ownership:
  - Kernel memory / objects: Owned by the kernel.
  - Extension memory / objects: Owned by the extension.
  - CPU: Owned by the extension.*

* Temporarily leased by the kernel to the extension for a given time quantum.

Heap

- Built on top of BPF arena.
  - Sparse memory region for extensions.
  - Pages can be allocated and deallocated in this region.
  - Surrounded by guard pages that trap accesses.
- Supports > 4 GB size.
- Uses a different sandboxing / SFI scheme.

Heap SFI

- Depends on aligned allocation of heap regions.
- Alignment is equal to size; always rounded up to a power of 2.
- Possibly wasted virtual address space, which is cheap.
- Rely on the verifier to reduce guard emission (range analysis etc., data flow).
- On x86: r12 == Heap Base (`0xdead0000`), r9 == Heap Mask (`0x0000ffff`).

Instrumentation →

```
r0 = heap_ptr;

r1 = *(u64 *)r0;
```

```
r0 = heap_ptr;
guard(r1);
r1 = *(u64 *)r0;
```

JIT →

```
mov rax, ...
and rax, r9
mov rdi, [r12+rax+0]
```

Translation

- Arenas / heaps can be shared with user space.
- Pointers escaping into them need to be translated.
- Rely on user space to map arena at address aligned to size.
- Uses addr_space_cast instruction.

- Translation from kernel to user:
  - mov, or, test, cmov
- Rely on verifier to know NULL-ness, can skip ‘test’.
- Translation from user to kernel:
  - Simple guard and preserve shared lower bits.

Cancellations

- Core idea of stack unwinding and resource cleanup in last year's LPC talk.

- An extension may have multiple **cancellation points (CPs)**.
  - E.g. accessing a not present page.
- For each CP, build a table of acquired kernel resources (through the verifier).
- At runtime, look up the table corresponding to the CP.
- Walk all entries, release resources.

Non-terminating loops

- Instrument back-edge with ***terminate.**
  - Load of a valid benign address.
  - It is loading from prog->aux->terminate_addr.
- Reset to NULL for triggering a page fault.
  - Marked as a cancellation point, triggers cancellation.
- Affects this extension on all CPUs, since terminate_addr is per-prog.
  - Policy decision: Cancel and force unload a bad extension ASAP.
  - As future work, can be isolated to a single CPU.

Example

```
void prog(struct xdp_md *ctx) {
    sk = bpf_sk_lookup(...);
    kflex_spin_lock(lock);            <------ CP-1
    while (node != NULL) {
        if (node->key == key)         <------ CP-2
            break;
        node = node->next;            <------ CP-3
    }                                 <------ CP-4
    kflex_spin_unlock(lock);
    bpf_sk_release(sk);
}
```

Object Table

sk	bpf_sk_release

Recovery

- Triggered through softlockup and hardlockup detection logic.
  - Granularity in order of seconds.
- If an extension is hung on a CPU, cancel it.
  - Set prog->aux_terminate_addr = 0;
  - Next time ***terminate** is accessed, program will incur a page fault.
  - Trigger cancellation from PF handler by inspecting program counter.

- Causes of lockup are loops instrumented with ***terminate.**

Co-designing extensions with user space

- Holding locks from both user-space and the kernel.
- Translation of pointers for bi-directional data access.
- Introduce bpf_preempt_{enable,disable}.
- MCS lock implemented in the extension over heaps.
- MCS nodes reside in the heap.

Two examples:

- Extension memory allocator.
- Memcached in XDP, with GC happening in user space.

Time Slice Extension

- User space may be preempted within a critical section.
- Set a bit in rseq region, and the scheduler will grant a one time extension.
- Configurable time quantum of extension.
- 50us-100us a reasonable value for short sections.

- If user space is hung or killed while holding the lock.
  - Extensions waiting and spinning will be eventually cancelled.
- If user space corrupts the lock.
  - Random memory corruption occurs, but only in the extension’s own memory.

Memory Allocator (malloc/free)

- Map heap in user space.
- Use jemalloc, replace mmap backend with heap (jemalloc arenas).
- User space thread prepares and refills per-CPU cache for extension.
- Use ringbuf to deliver wake up when running low on memory.

Benefits

- Extension side is simply a per-CPU cache, and shared pool with user space.
- Fragmentation handled by jemalloc.

Downsides

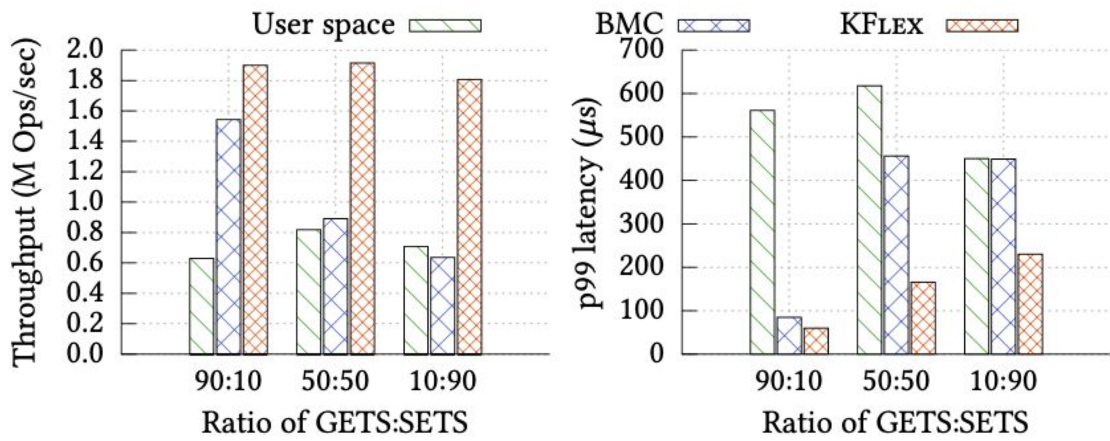
- OOM in extension if user-space thread is killed.

Performance Mode

- Elide guards when heap pointer is read.
- Trades off confidentiality for performance.
- Extensions may read arbitrary memory.
- Reading user addresses leads to page fault, which leads to cancellation.

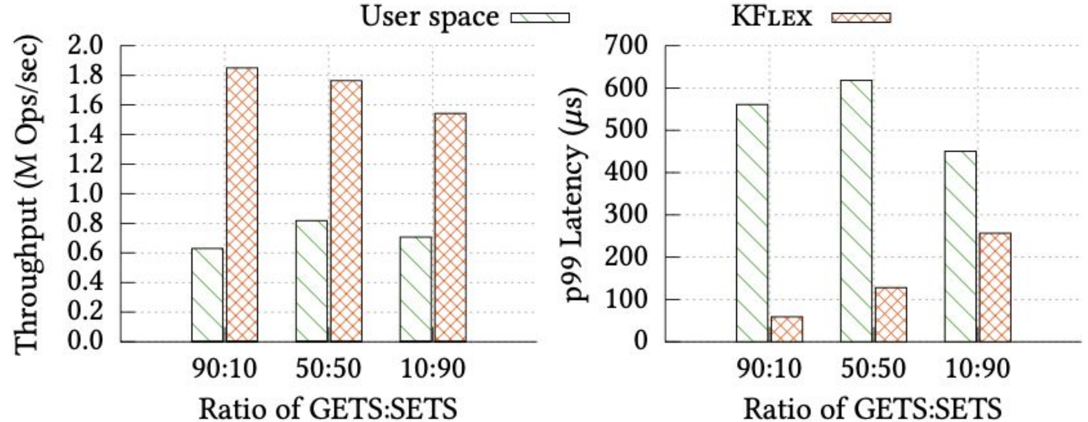
Evaluation

Memcached in XDP



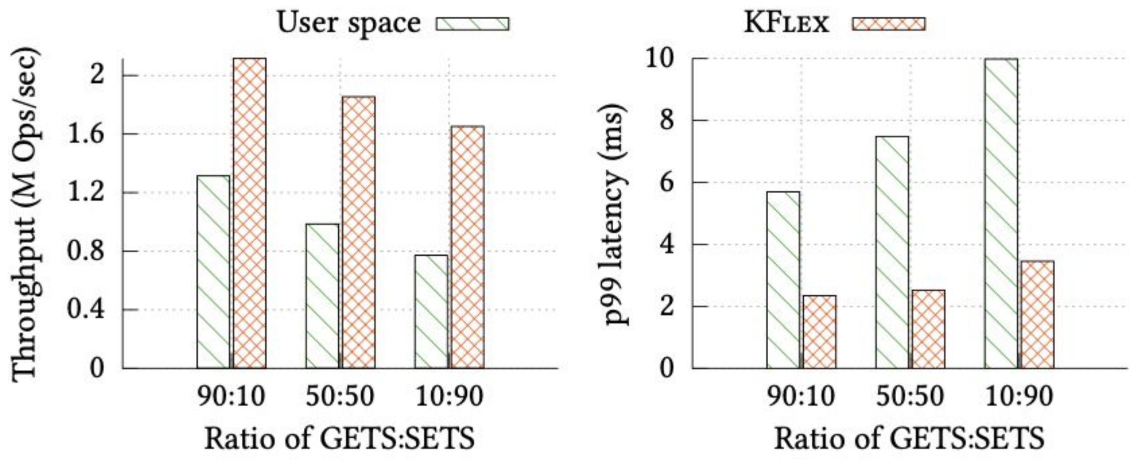
Up to 2.8x and 3x more throughput than BMC and user space Memcached. Up to 1.9x and 9x reduction in p99 latency, resp.

Memcached in XDP, with GC in user space



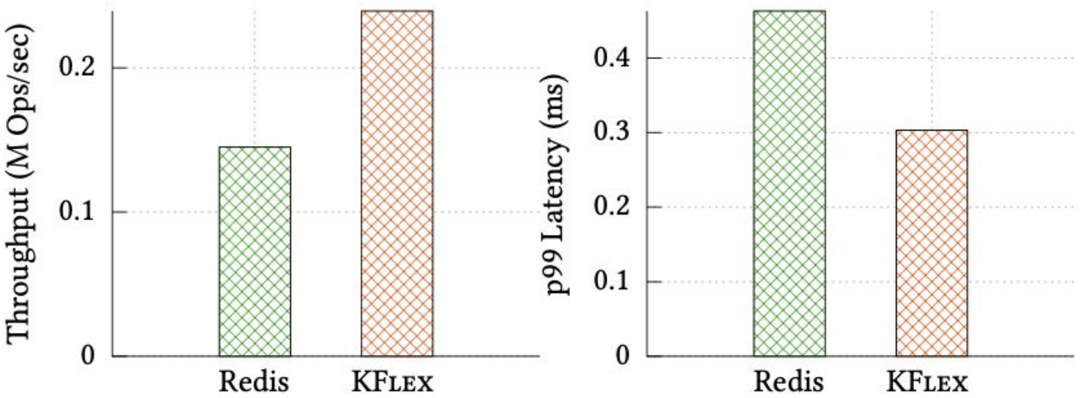
2.2x-2.9x improvement in throughput. 42%-90% reduction in p99 latency across the three ratios.

Redis in sk_skb - GETS/SETS



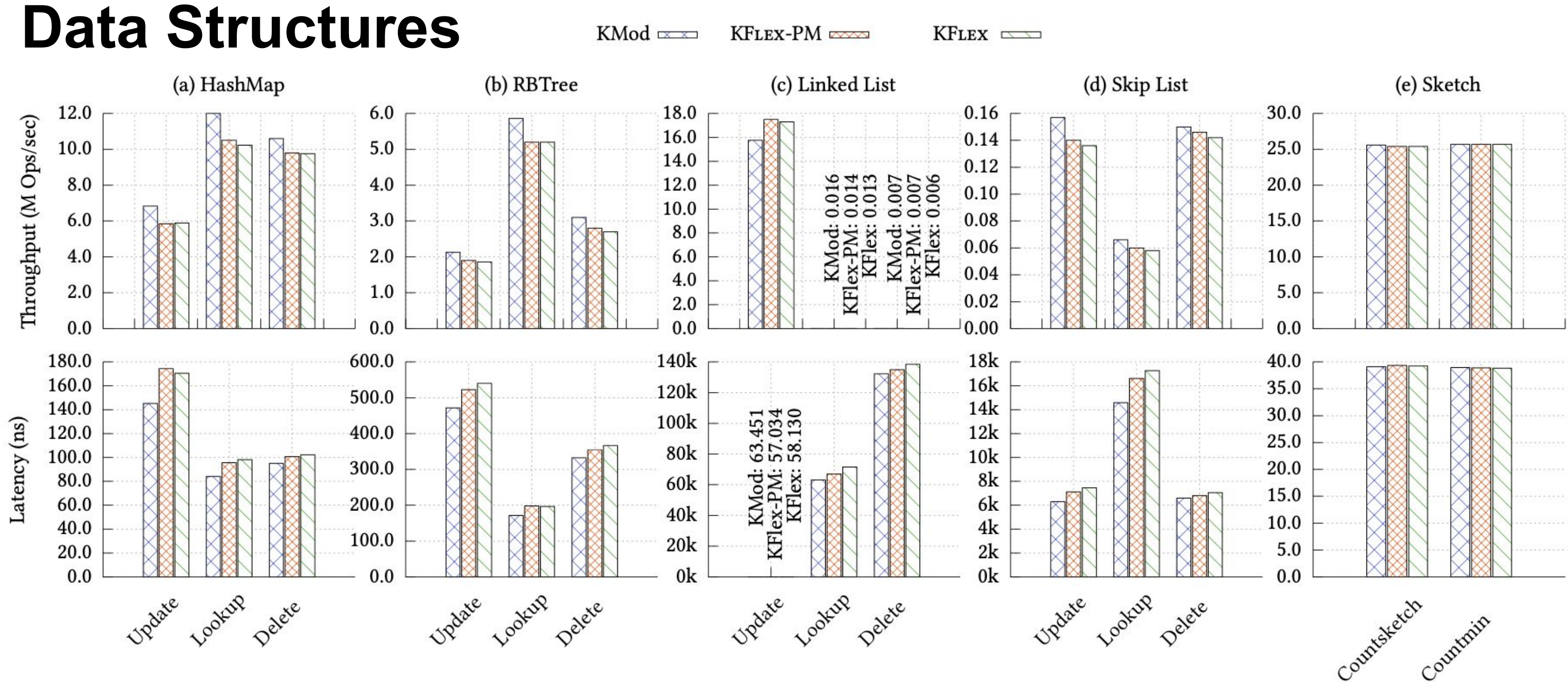
Up to 2x more throughput, and up to 3x lower p99 latency.

Redis in sk_skb - ZADD



1.6x more throughput than user space. 30% reduction in p99 latency.

Data Structures



On average, 7% throughput overhead (max 18%), 30% latency overhead, when compared against kernel implementation.

Integrating verification + SFI

- SFI relies on guard pages, and the verifier's range analysis (var_off tracking).
- Guards need to be emitted upon pointer modification (to sanitize address).
- On average, 3/4th can be elided using known information in either reg->off or var_off tnum.

Wishlist

- Sandboxing using protection keys.
  - PKS appears to be dead, MPK with U bit set for arena pages (?).
  - Domain switching cost.
- Growing register count for eBPF.
  - More register pressure, not as bad on x86, but could be significant on ARM.
- Mixing kernel and arena objects.
  - Pointers read from arena objects can’t be trusted.
  - Something like an FD table, but for kernel objects? May not scale.
- Locking (WIP).
  - Using MCS locks implemented in BPF itself.
  - Not portable; no BPF memory model (yet).
  - Upon deadlocks, relies on watchdog-driven cancellation to abort program.

Questions?

For more details, check the paper.
https://rs3lab.github.io/KFlex

Prior Art

- bpf_obj_new, BPF Linked List, BPF RB-Tree
- Helpers used to manipulate linked lists or red-black trees.
- Preserve invariants, represent ownership, account for object lifecycle.

Downsides

- Mostly verifier complexity; reason about aliasing, object lifetimes, ownership.
- Since we hand kmalloc memory to program, we must release it back.
- Drive-by growth: New use case, new support.
- Maybe not sustainable in the long run?