Fast, Flexible, and Practical Kernel Extensions

Kumar Kartikeya Dwivedi, Rishabh Iyer, Sanidhya Kashyap

Kernel extensions are critical

- Mechanism to safely modify the kernel at runtime

Kernel extensions are critical

- Mechanism to safely modify the kernel at runtime

Kernel extensions are critical

- Mechanism to safely modify the kernel at runtime
- Used for observability, security, networking

Kernel extensions are critical

- Mechanism to safely modify the kernel at runtime
- Used for observability, security, networking
- Emerging use cases: Application offloads, CPU scheduling

eBPF Katran cilium

Kernel extensions are critical

- Mechanism to safely modify the kernel at runtime
- Used for observability, security, networking
- Emerging use cases: Application offloads, CPU scheduling
- eBPF is 1% of all CPU cycles globally on Meta's fleet

eBPF Katran cilium

Ideal extensibility goals

Safety: Cannot crash or stall the kernel

Flexibility: Allow diverse behavior in extension code

Performance: Low overhead on execution

Practicality: Language-independence

Ideal extensibility goals

Safety: Cannot crash or stall the kernel

Flexibility: All

Performance:

Practicality: L

Ideal extensibility goals

Safety: Cannot crash or stall the kernel

Flexibility: Allow diverse behavior in extension code

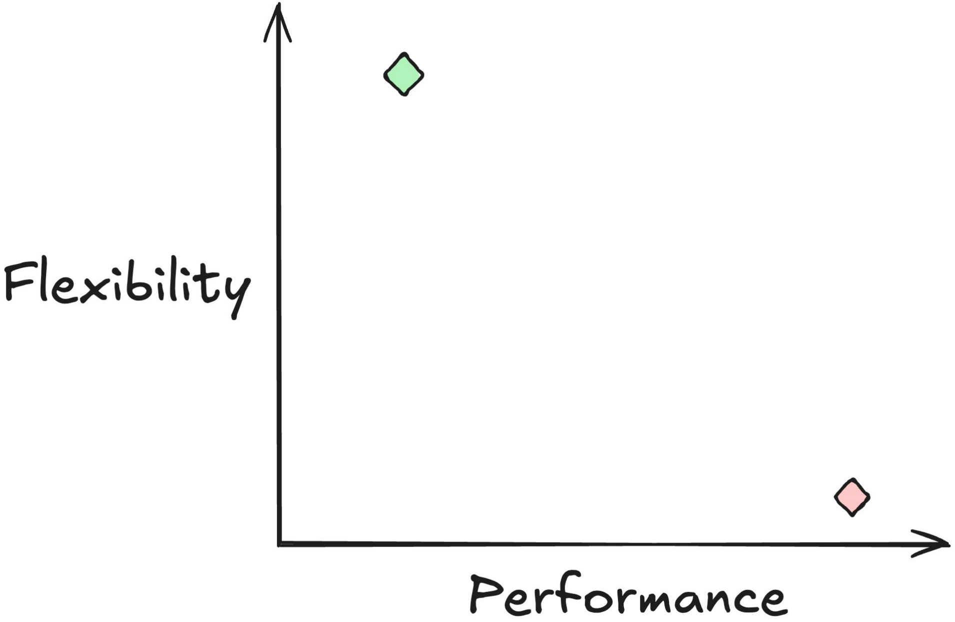
Performance: Low overhead on execution

Practicality: Language-independence

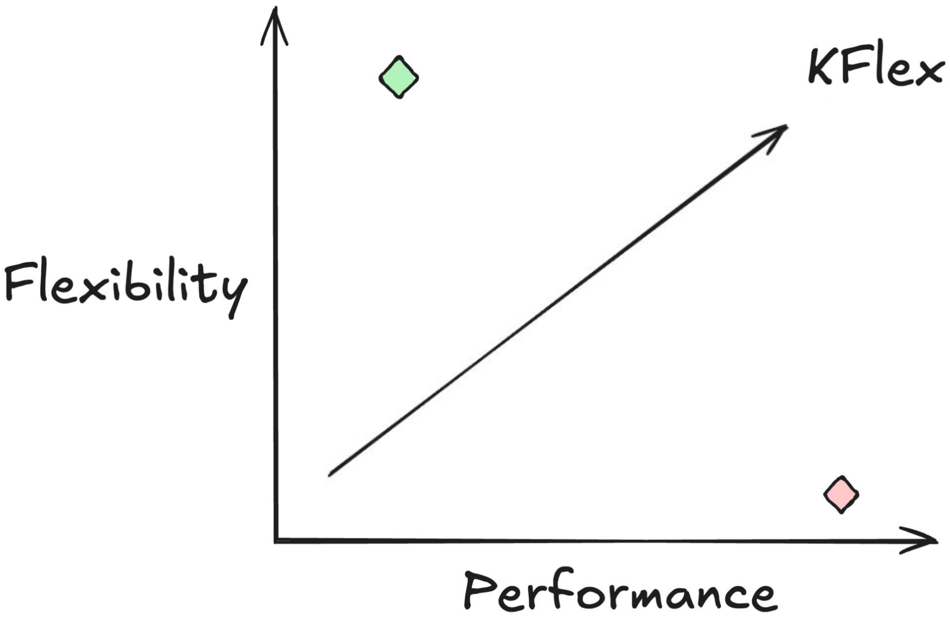
Safety is fundamental for kernel extensions

Problem Statement

Kernel extensibility today is either flexible or performant — not both

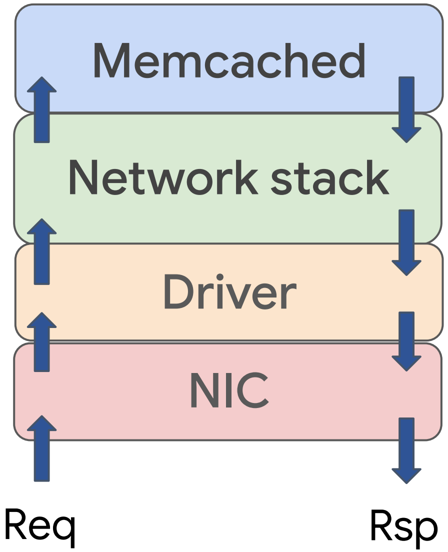


KFlex: fast, flexible, and practical extension framework



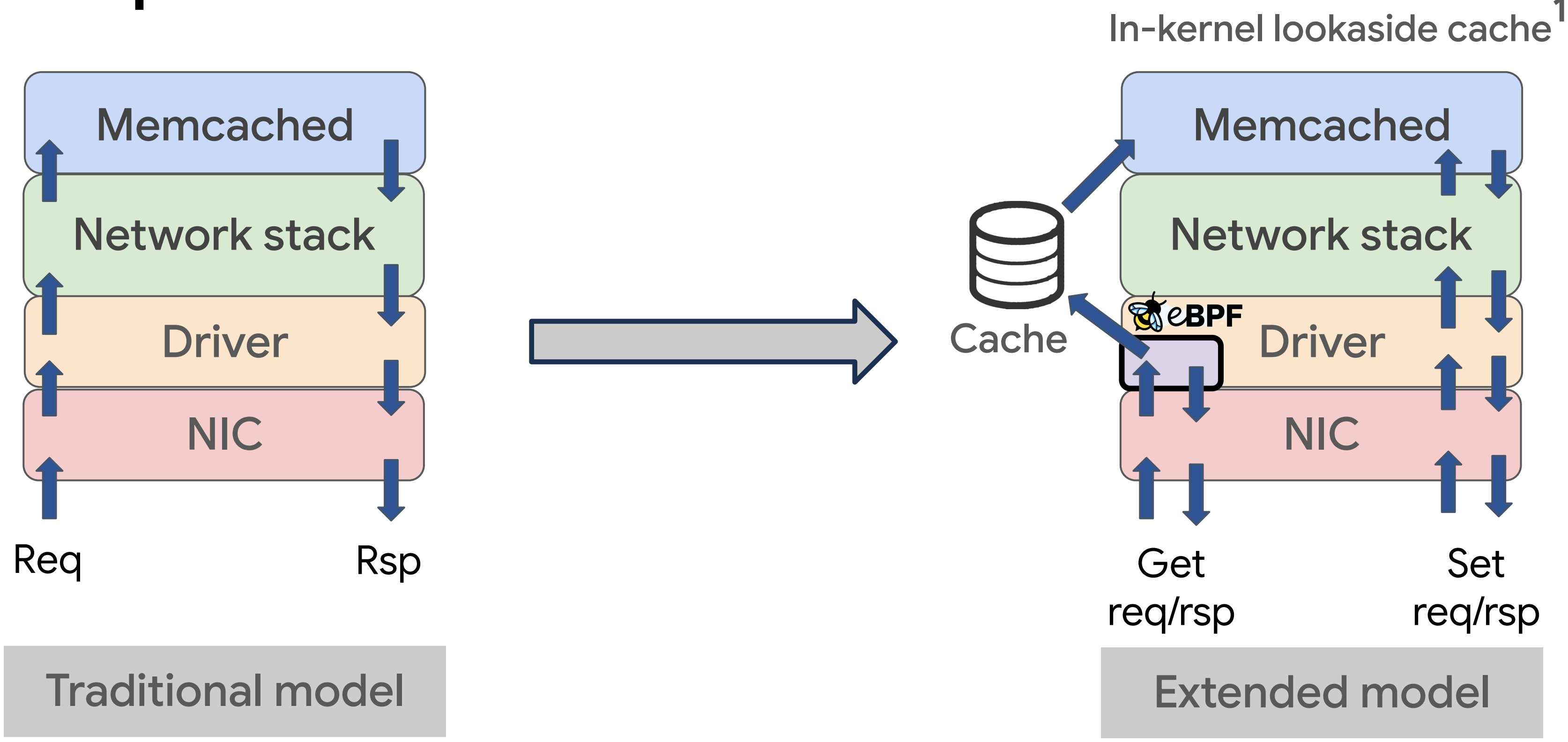
Upstreamed into the Linux kernel mainline

Example use case: Memcached offload



Traditional model

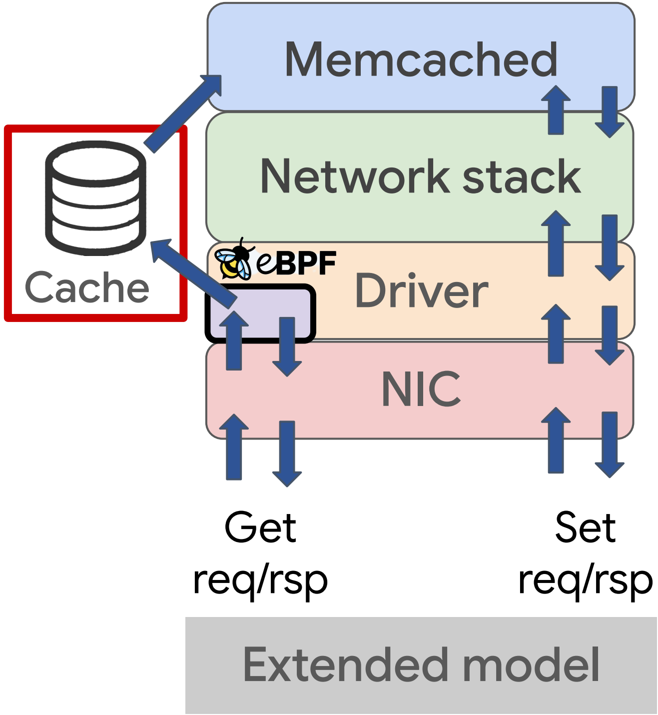
Example use case: Memcached offload



[1] BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing, NSDI'21

Lack of flexibility with eBPF

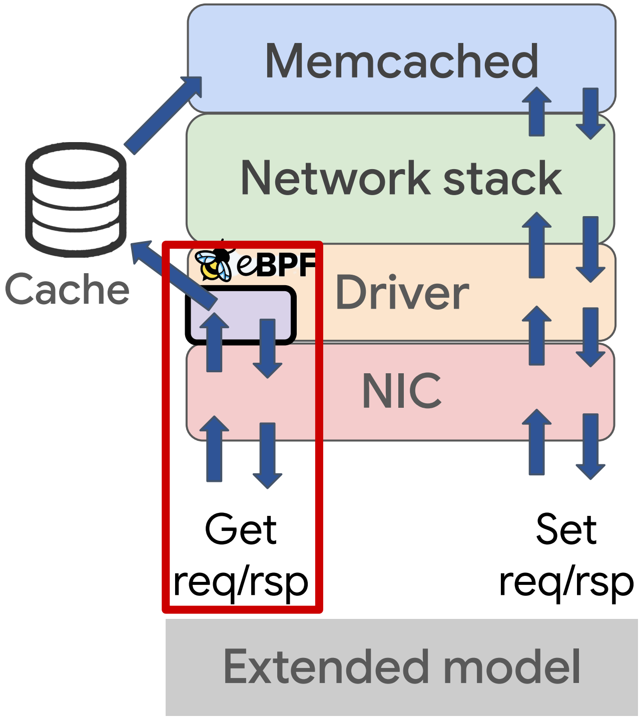
- Data structures cannot be shared
  - Wasted memory



[1] BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing, NSDI'21

Lack of flexibility with eBPF

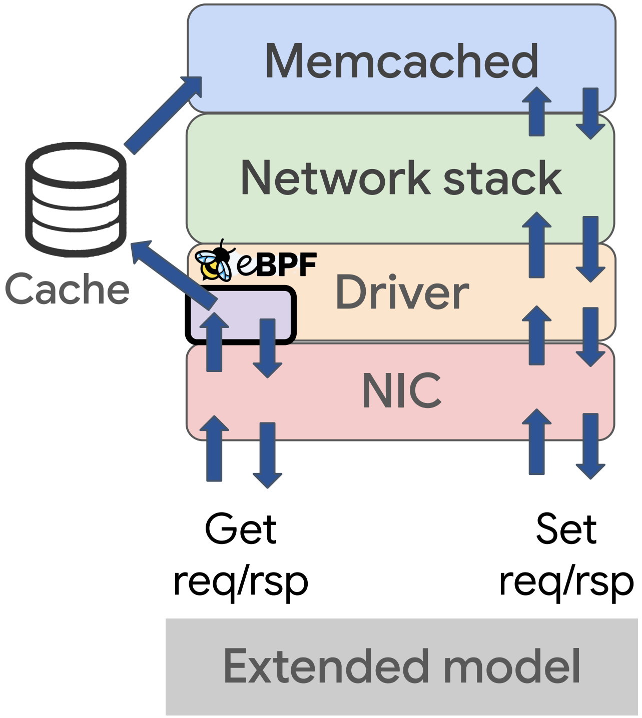
- Data structures cannot be shared
  - Wasted memory
- No memory allocation
  - Only handle GETs



[1] BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing, NSDI'21

Lack of flexibility with eBPF

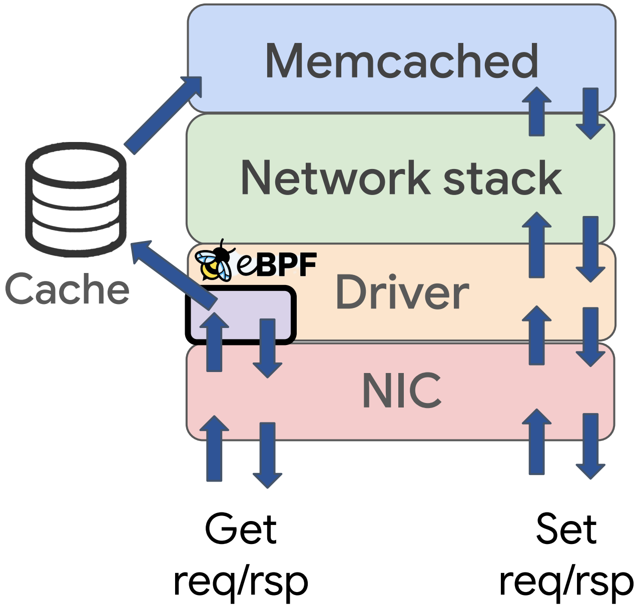
- Data structures cannot be shared
  - Wasted memory
- No memory allocation
  - Only handle GETs
- No user-defined data structures



[1] BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing, NSDI'21

Lack of flexibility with eBPF

- Data structures cannot be shared
  - Wasted memory
- No memory allocation
  - Only handle GETs
- No user-defined data structures



Current extensibility approach to safety hurts flexibility

eBPF overview: linked list iteration

```
struct list_head *head;
int prog(struct xdp_md *ctx) {
    while (head != NULL) {
        head = head->next;
    }
    return bpf_redirect(...);
}
```

Linked list head

eBPF overview: linked list iteration

```
struct list_head *head;
int prog(struct xdp_md *ctx) {
    while (head != NULL) {
        head = head->next;
    }
    return bpf_redirect(...);
}
```

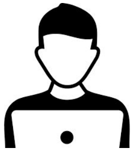
Linked list head

Linked list iteration

eBPF overview

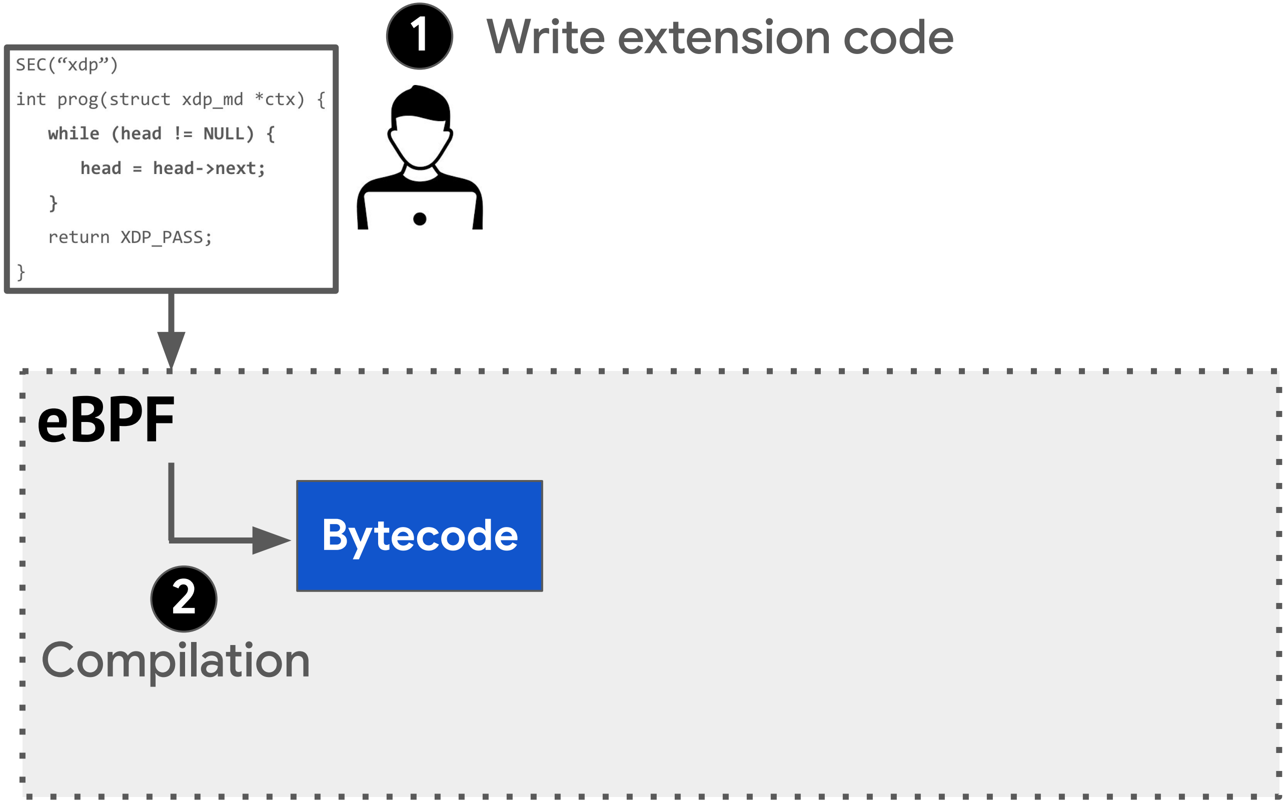
1 Write extension code

```
SEC("xdp")
int prog(struct xdp_md *ctx) {
   while (head != NULL) {
      head = head->next;
   }
   return XDP_PASS;
}
```

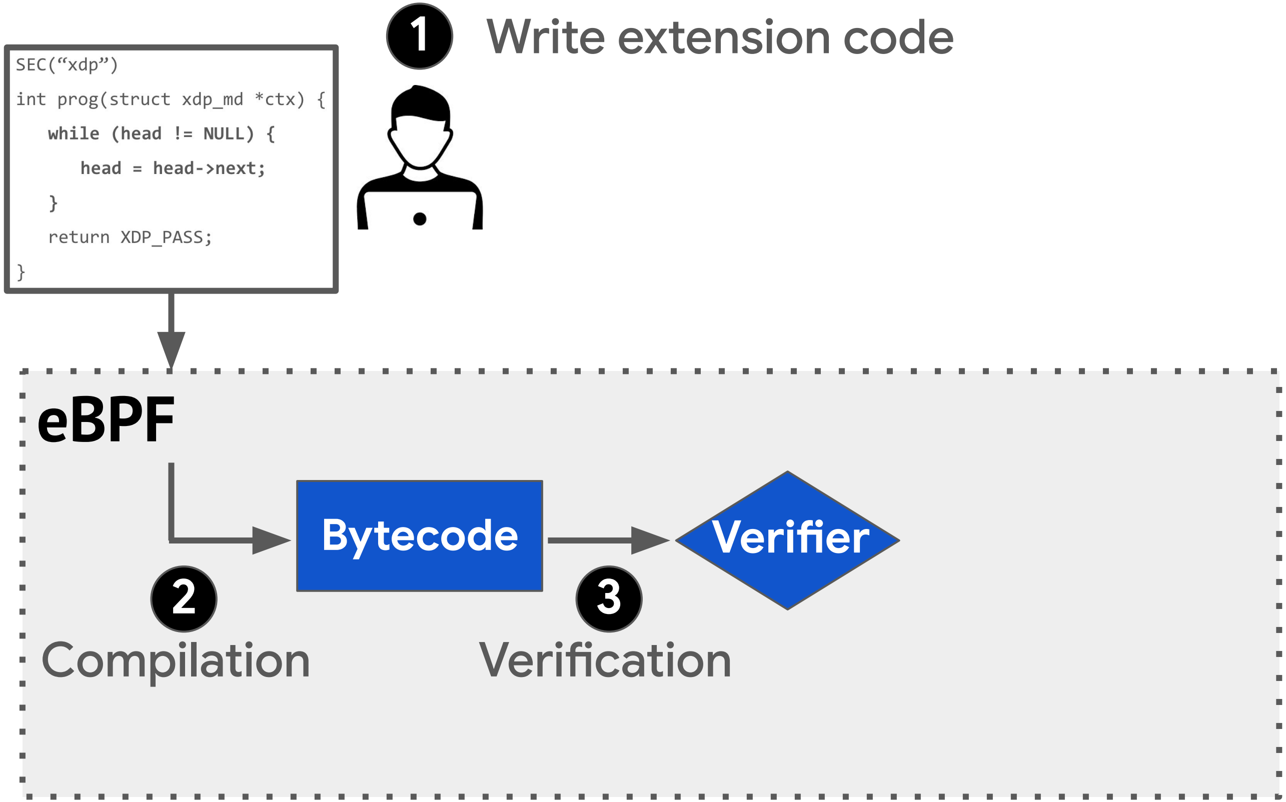


eBPF

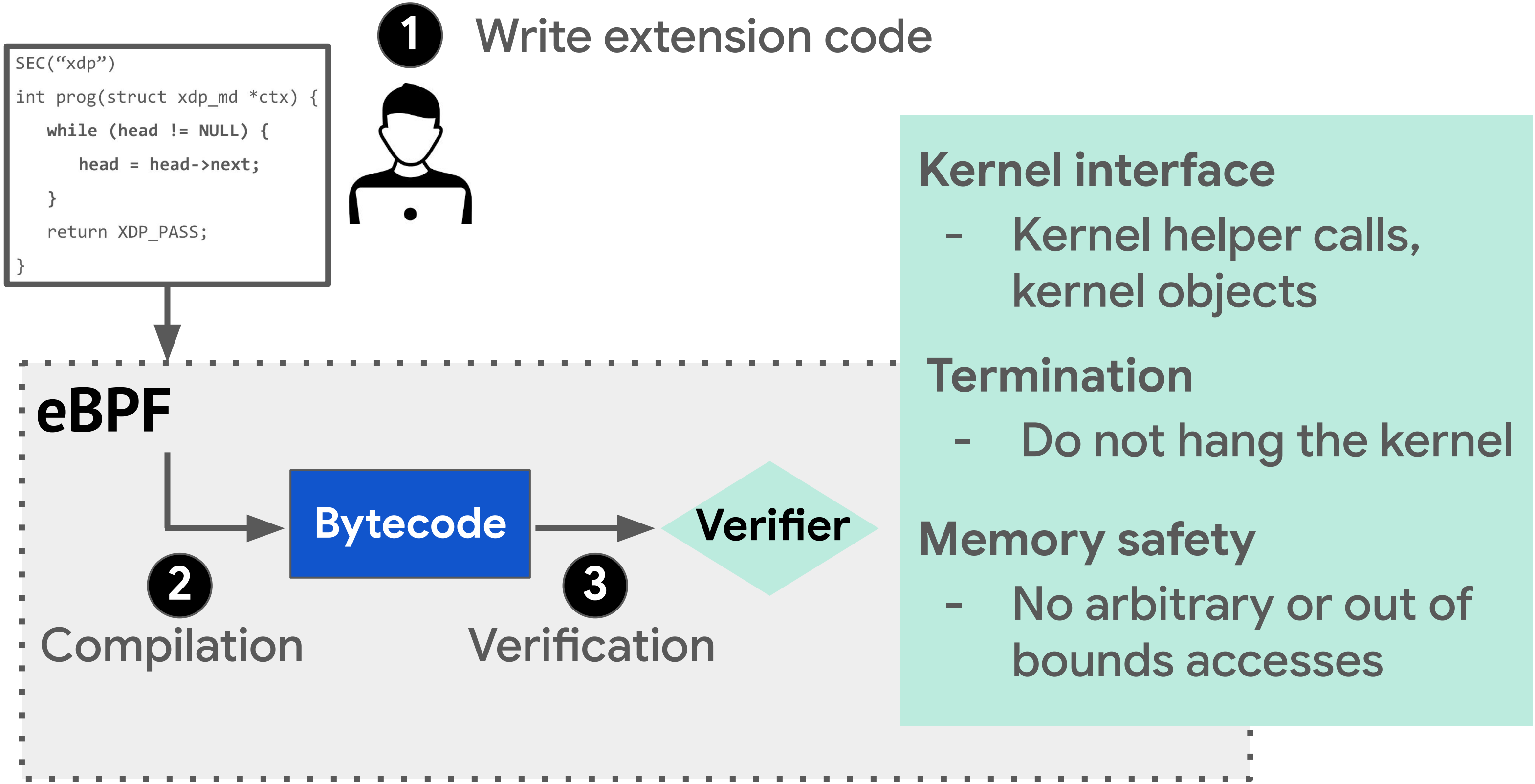
eBPF overview



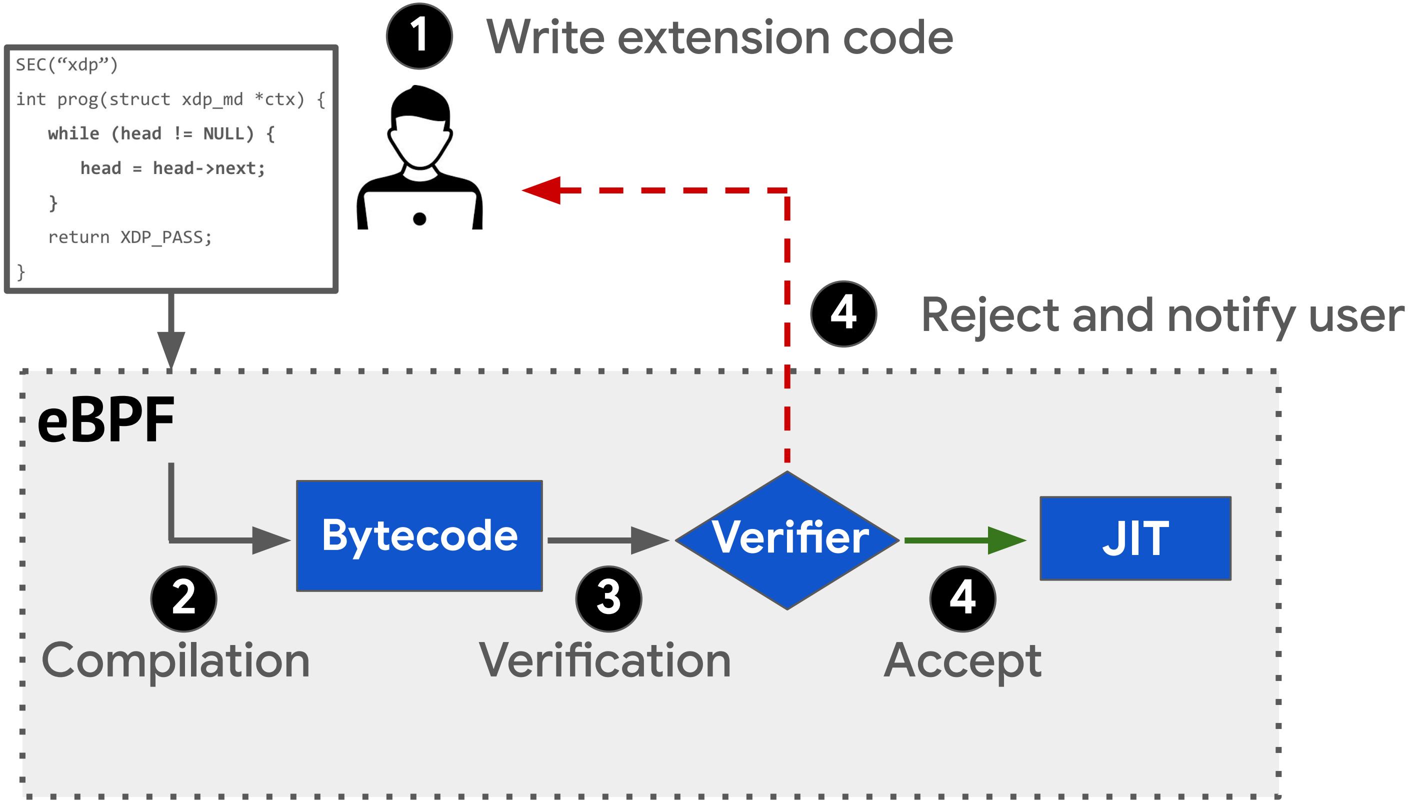
eBPF overview



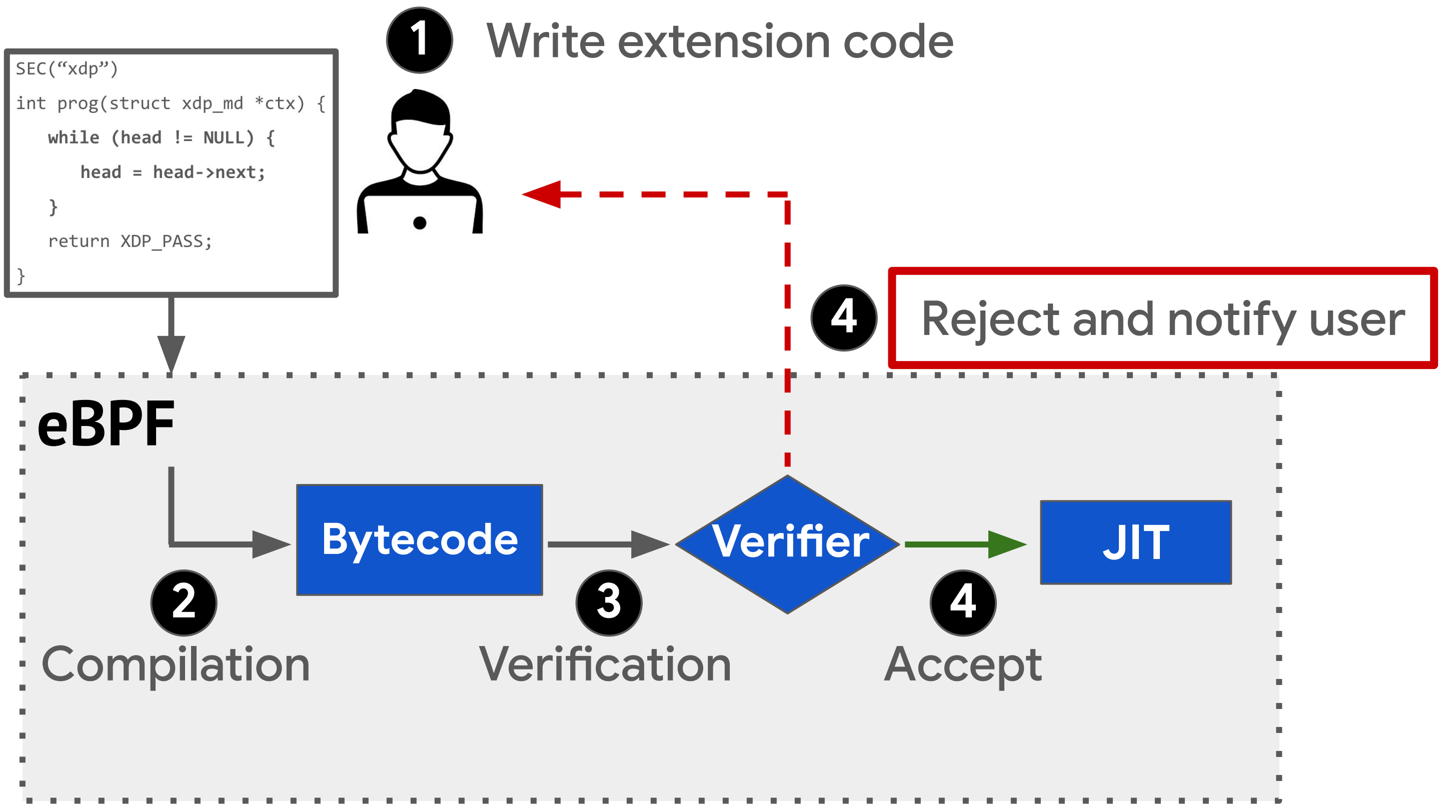
eBPF overview



eBPF overview



eBPF overview



eBPF: issues with current design

```
int prog(struct xdp_md *ctx) {
    while (head != NULL) {
        head = head->next;
    }
    return bpf_redirect(...);
}
```

Verifier

Kernel interface

- Kernel helper calls, kernel objects

Termination

- Do not hang the kernel

Memory safety

- No arbitrary or out of bounds accesses

eBPF: issues with current design

```
int prog(struct xdp_md *ctx) {
    while (head != NULL) {
        head = head->next;
    }
    return bpf_redirect(...);
}
```

Verifier

Kernel interface
- Kernel helper calls, kernel objects

Termination
- Do not hang the kernel

Memory safety
- No arbitrary or out of bounds accesses

eBPF: safety of kernel interfaces

```
int prog(struct xdp_md *ctx) {
    while (head != NULL) {
        head = head->next;
    }
    return bpf_redirect(...);
}
```

Verifier

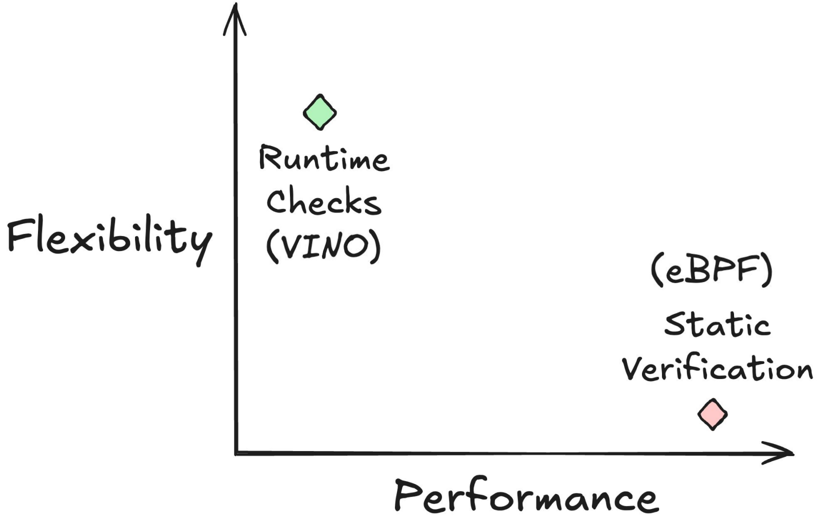
Kernel interface
- Kernel helper calls, kernel objects

Termination
- Do not hang the kernel

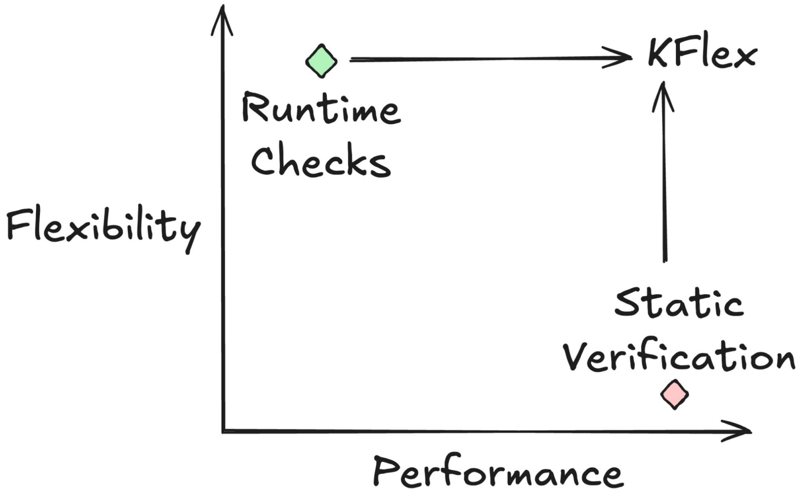
Memory safety
- No arbitrary or out of bounds accesses

Problem statement

Kernel extensibility is either flexible, or performant — not both



KFlex



An extension framework for arbitrary code extensibility

Insight: separate safety properties

Kernel helper calls, kernel-owned memory

Kernel interface compliance

Kernel interface
- Kernel helper calls, kernel objects

Termination
- Do not hang the kernel

Memory safety
- No arbitrary or out of bounds accesses

Insight: separate safety properties

Kernel helper calls, kernel-owned memory

Kernel interface compliance

Flexibility is w.r.t extension memory & time

Extension correctness

Kernel interface

- Kernel helper calls, kernel objects

Termination

- Do not hang the kernel

Memory safety

- No arbitrary or out of bounds accesses

KFlex: use dedicated mechanisms

- Kernel interface compliance: Narrow, well-defined

Static verification

KFlex: use dedicated mechanisms

- Kernel interface compliance: Narrow, well-defined

Static verification

- Extension correctness: Diverse and arbitrary behavior

Runtime checks

KFlex: use dedicated mechanisms

- Kernel interface compliance: Narrow, well-defined

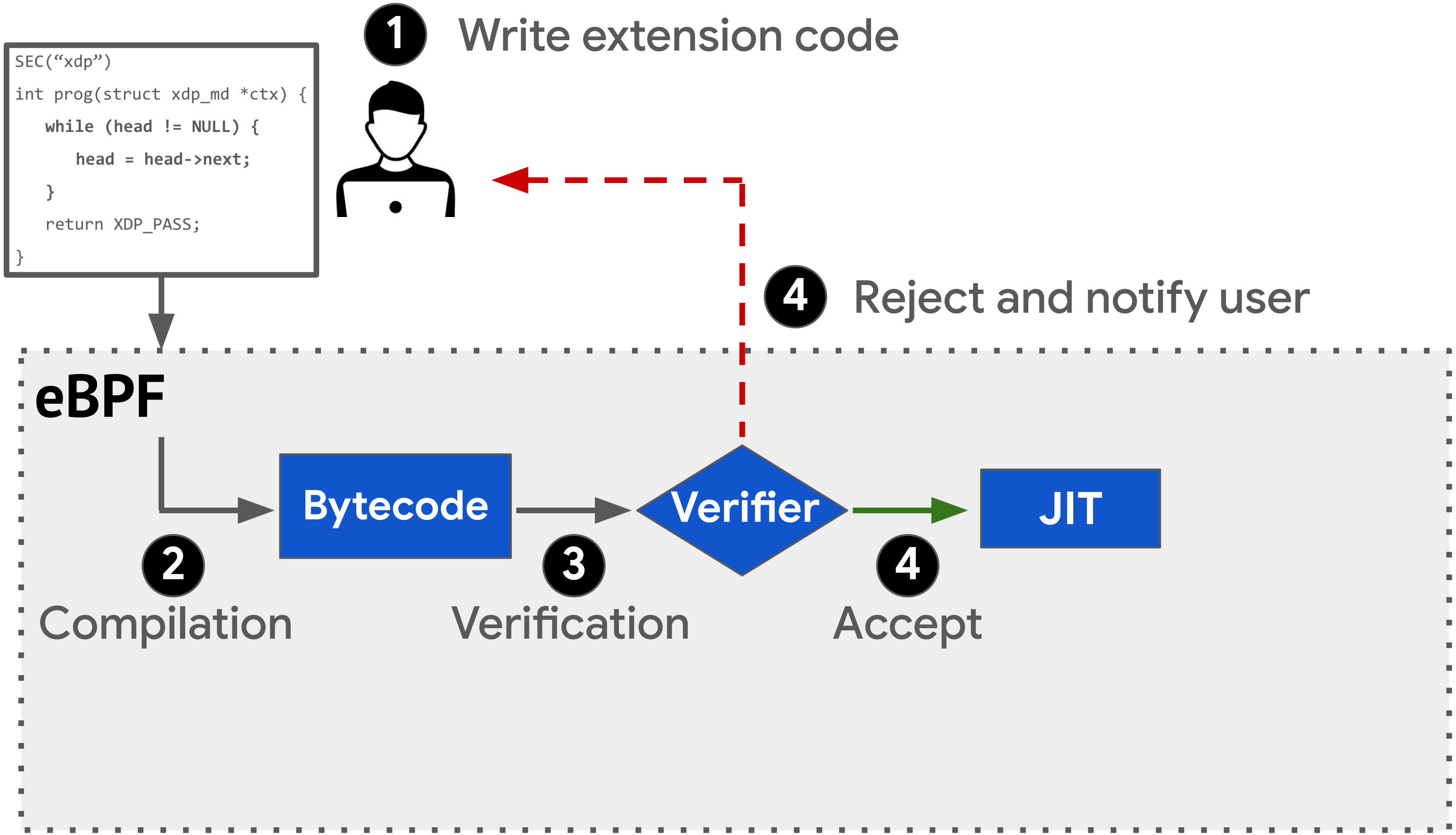
Static verification

- Extension correctness: Diverse and arbitrary behavior

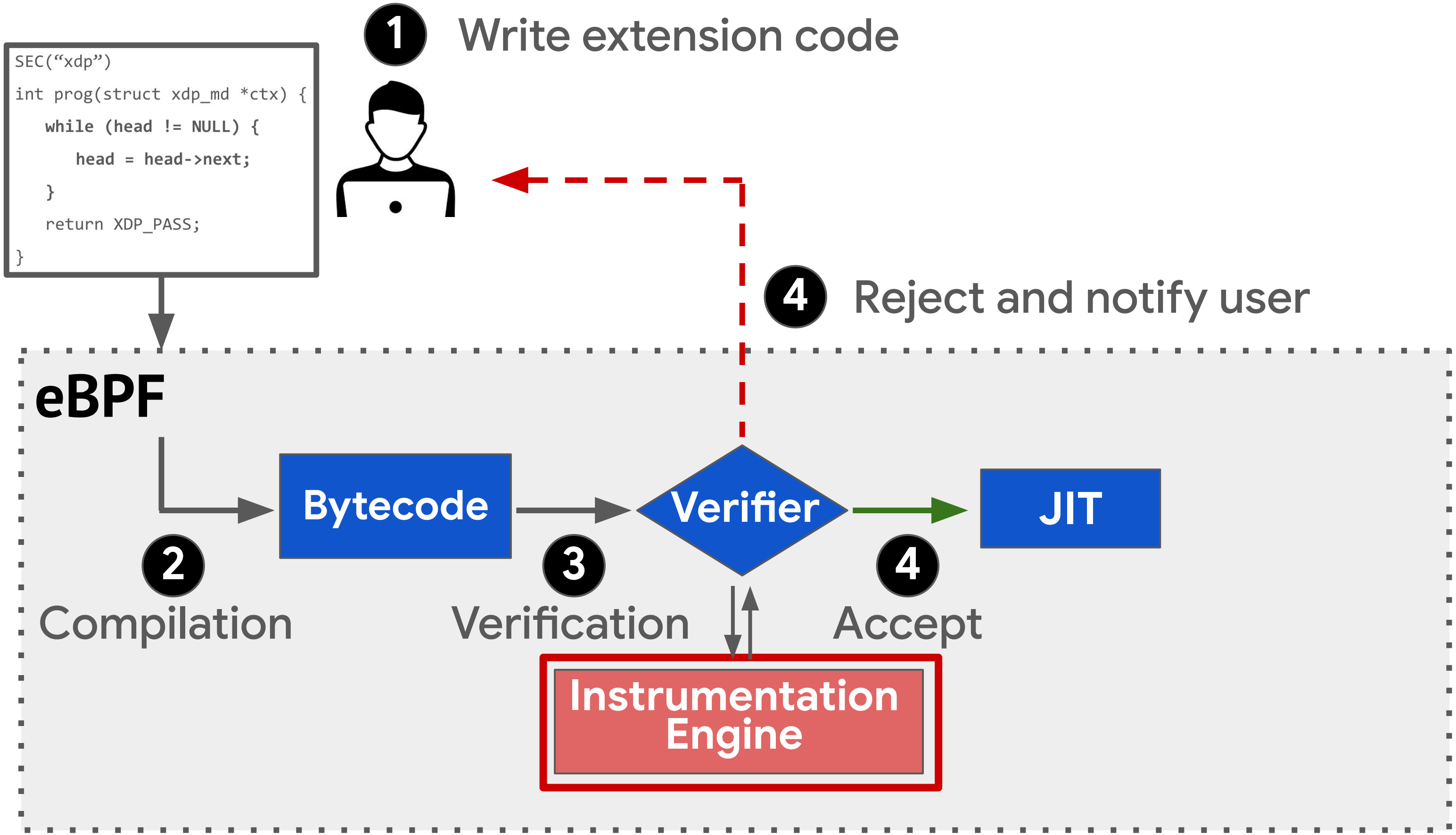
Runtime checks

Eliminate runtime overhead with co-design of runtime checks and verification

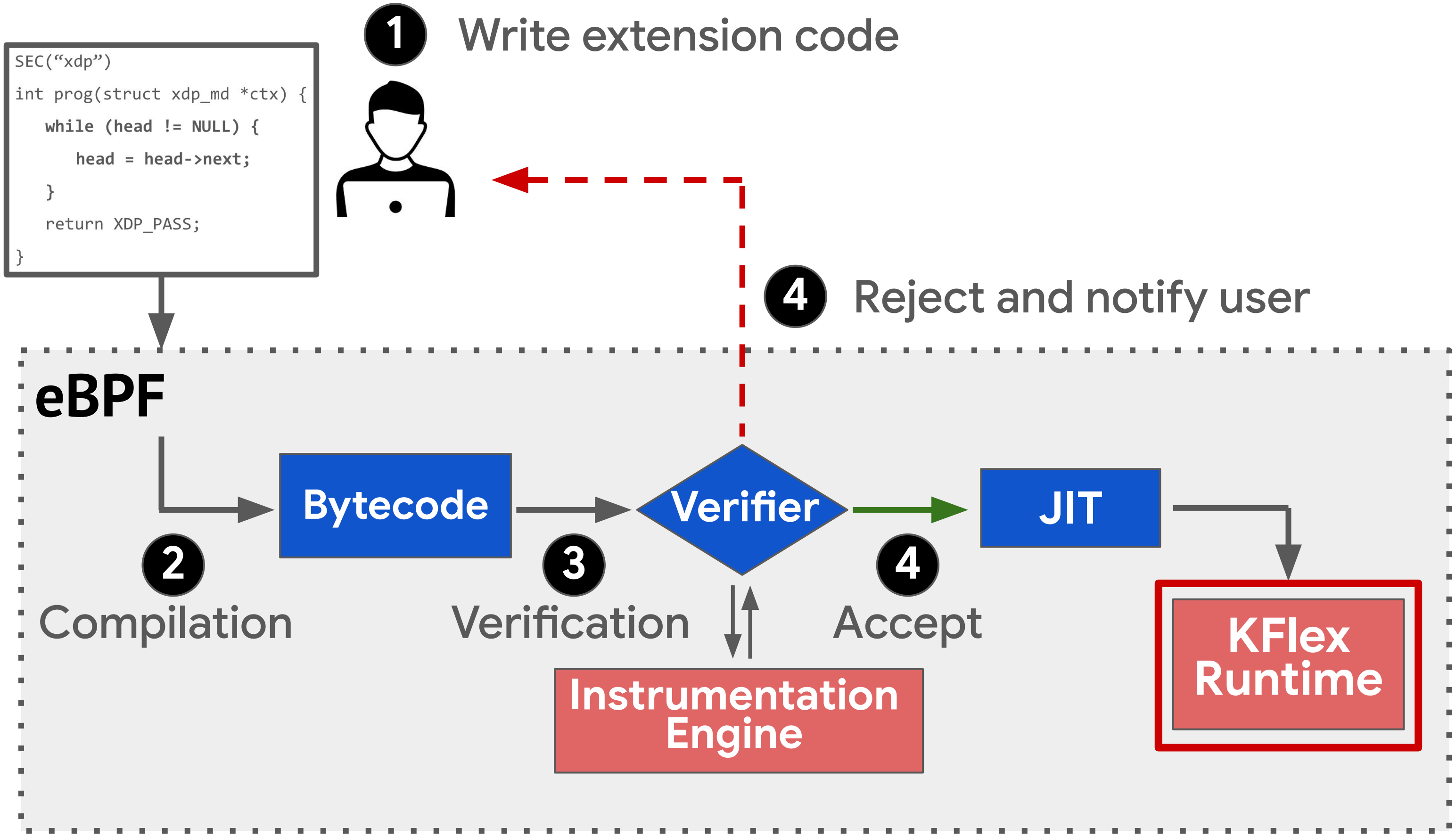
KFlex overview



KFlex overview



KFlex overview



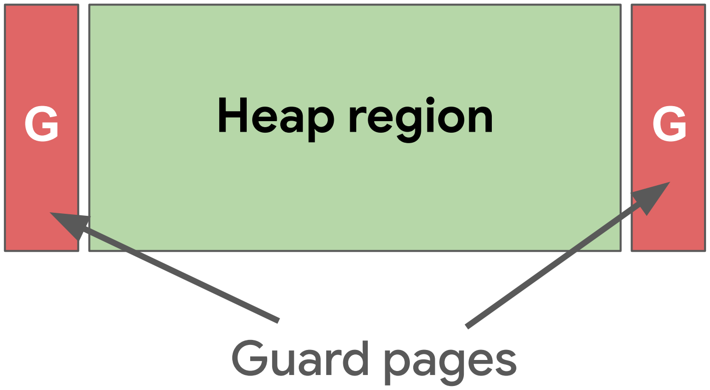
Extension correctness with runtime checks

- Memory safety for extension-owned data
- Safe termination to ensure forward progress

Memory safety using sandboxing

Dedicated region for extension-owned memory

- All extension data lives in heap
- Pages can be allocated and deallocated on demand
- Surrounded by guard pages that trap out-of-bounds accesses



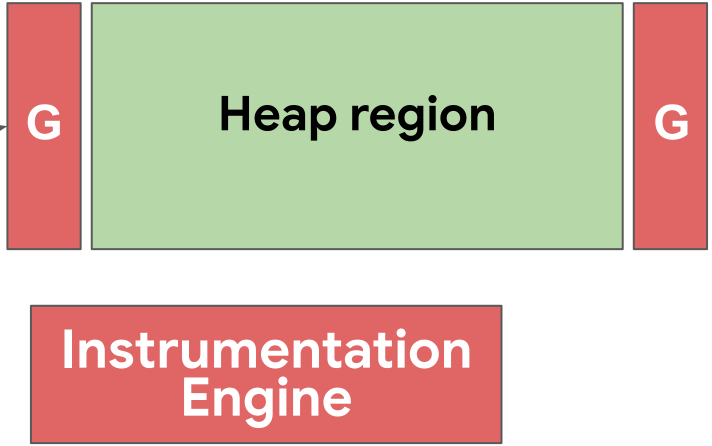
Memory safety using sandboxing

```
int prog(struct xdp_md *ctx) {
    while (head != NULL) {
        head = head->next;
    }
    return bpf_redirect(...);
}
```

May be out of bounds

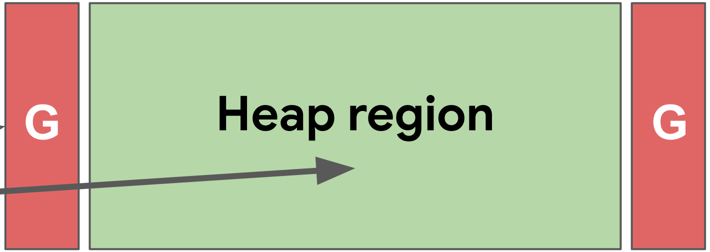
Memory safety using sandboxing

```
int prog(struct xdp_md *ctx) {
    while (head != NULL) {
        sanitize(head);
        head = head->next;
    }
    return bpf_redirect(...);
}
```



Memory safety using sandboxing

```
int prog(struct xdp_md *ctx) {
    while (head != NULL) {
        sanitize(head);
        head = head->next;
    }
    return bpf_redirect(...);
}
```



Within bounds!

Extension cancellations

- Safely terminate an extension at a given point in bounded time

Safe termination using extension cancellations

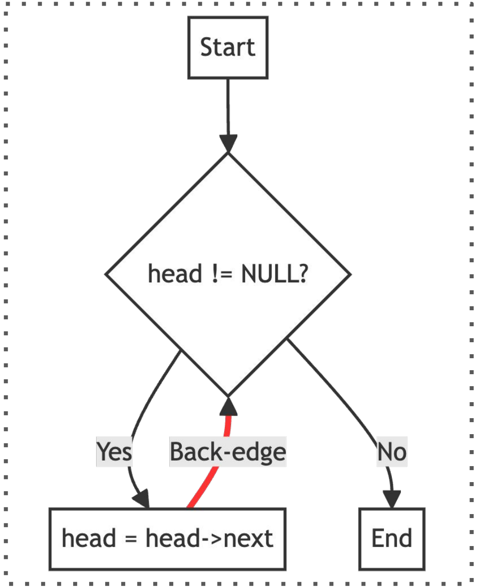
- Find non-terminating loops

```
while (head != NULL) {
   head = head->next;
}
```

Safe termination using extension cancellations

- Find non-terminating loops
- Instrument loop back-edges

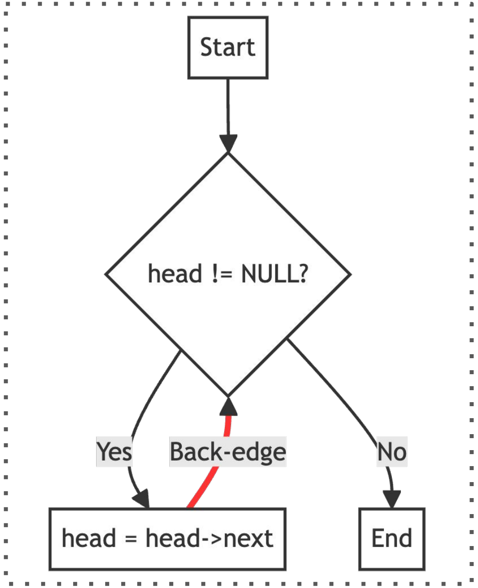
```
while (head != NULL) {
    head = head->next;
}
```



Safe termination using extension cancellations

- Find non-terminating loops
- Instrument loop back-edges
- Terminate and release kernel resources on a stall

```
while (head != NULL) {
    head = head->next;
}
```



Safe termination using extension cancellations

```
void prog(struct xdp_md *ctx) {
    sk = bpf_sk_lookup(...);
    while (head != NULL) {
        sanitize(head);
        head = head->next;


    }
    bpf_sk_release(sk);
    return bpf_redirect(...);
}
```

Safe termination using extension cancellations

```
void prog(struct xdp_md *ctx) {
    sk = bpf_sk_lookup(...);
    while (head != NULL) {
        sanitize(head);
        head = head->next;
        *terminate;
    }
    bpf_sk_release(sk);
    return bpf_redirect(...);
}
```

Instrumentation Engine

Safe termination using extension cancellations

```
void prog(struct xdp_md *ctx) {
    sk = bpf_sk_lookup(...);
    while (head != NULL) {
        sanitize(head);
        head = head->next;
        *terminate;
    }
    bpf_sk_release(sk);
    return bpf_redirect(...);
}
```

P1

Object Table

sk	bpf_sk_release

Recovery of the kernel

```
void prog(struct xdp_md *ctx) {
    sk = bpf_sk_lookup(...);
    while (head != NULL) {
        sanitize(head);
        head = head->next;
        *terminate;
    }
    bpf_sk_release(sk);
    return bpf_redirect(...);
}
```

Recovery of the kernel

```
void prog(struct xdp_md *ctx) {
    sk = bpf_sk_lookup(...);
    while (head != NULL) {
        sanitize(head);
        head = head->next;
        *(NULL);
    }
    bpf_sk_release(sk);
    return bpf_redirect(...);
}
```

Reset to NULL

KFlex Runtime

Recovery of the kernel

```
void prog(struct xdp_md *ctx) {
    sk = bpf_sk_lookup(...);
    while (head != NULL) {
        sanitize(head);
        head = head->next;
        *(NULL);
    }
    bpf_sk_release(sk);
    return bpf_redirect(...);
}
```

Page fault!

Recovery of the kernel

```c
void prog(struct xdp_md *ctx) {
    sk = bpf_sk_lookup(...);
    while (head != NULL) {
        sanitize(head);
        head = head->next;
        *(NULL);
    }
    bpf_sk_release(sk);
    return bpf_redirect(...);
}
```

P1

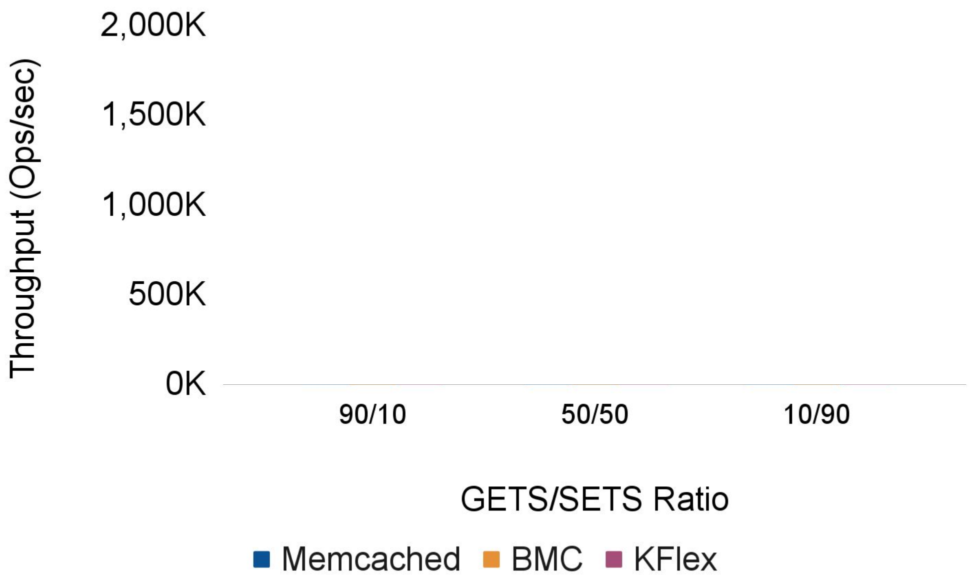
Object Table

sk	bpf_sk_release

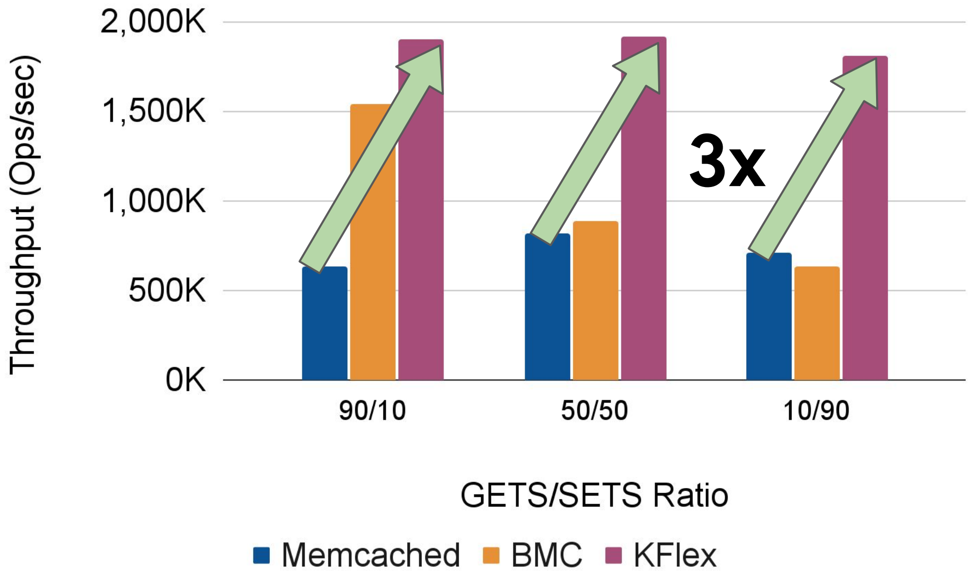
Evaluation

- Can KFlex improve end-to-end performance for applications?
- Can KFlex enable flexibility with low overhead?

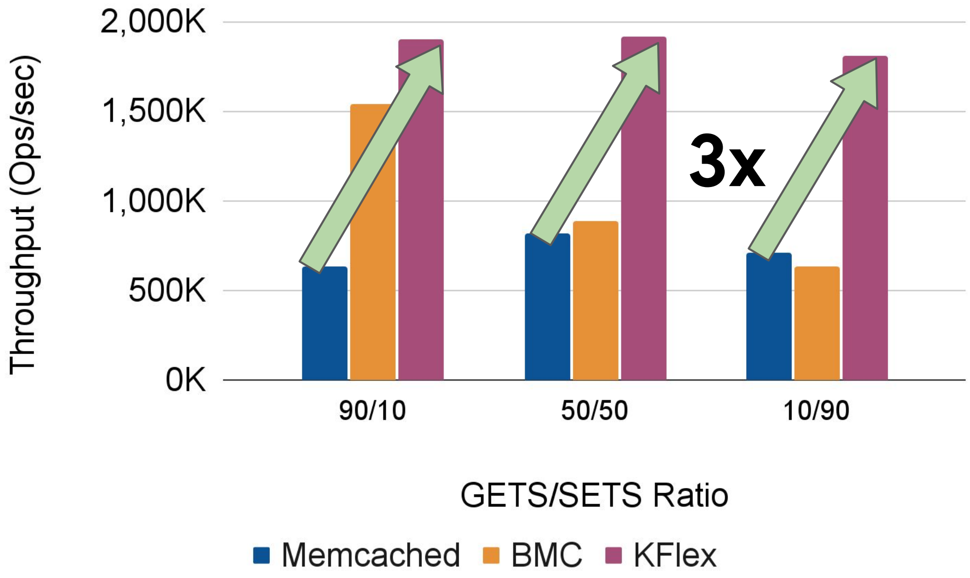
Memcached in XDP



Memcached in XDP



Memcached in XDP

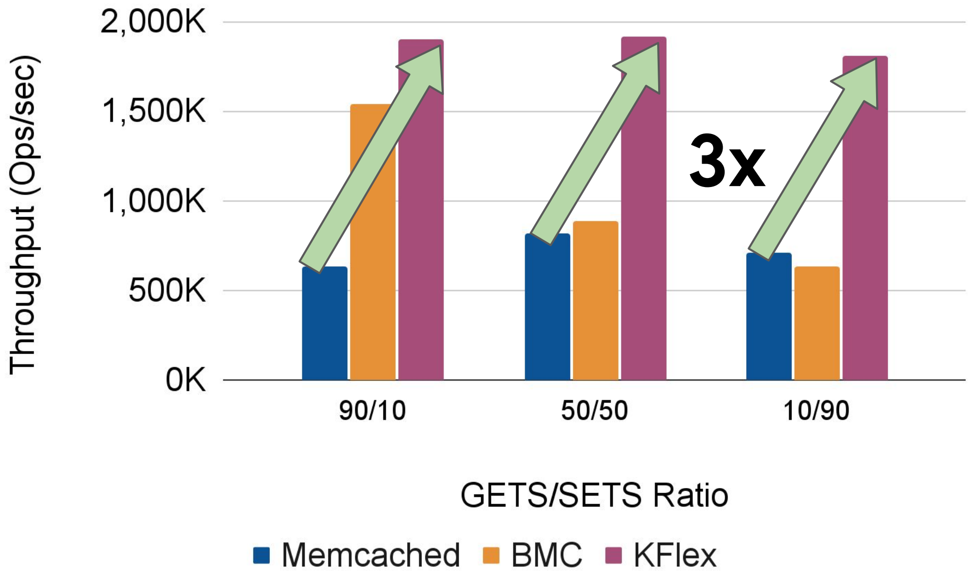


Allows both SETS/GETS

No memory waste

Low overhead

Memcached in XDP



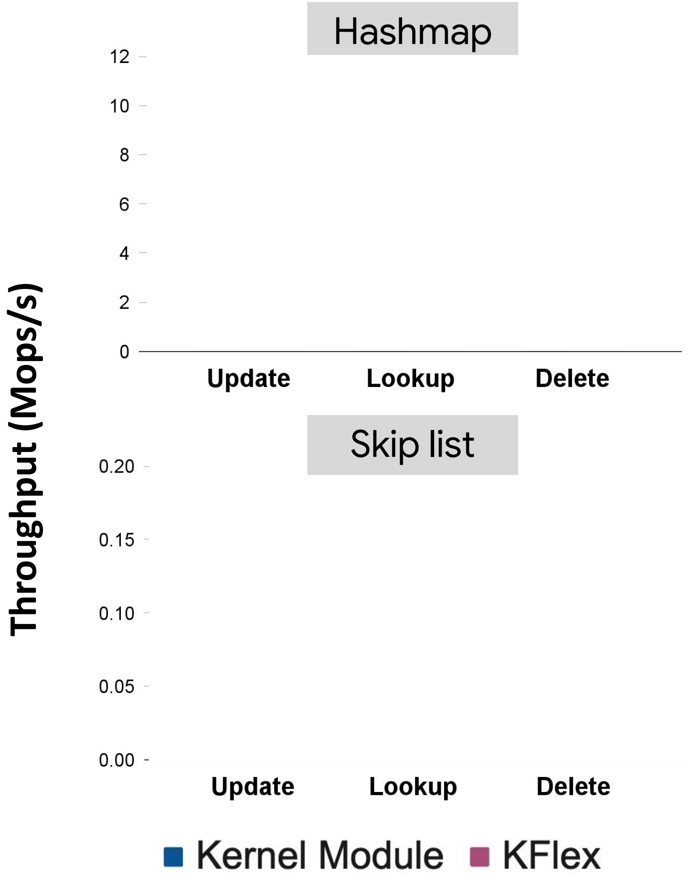
Allows both SETS/GETS

No memory waste

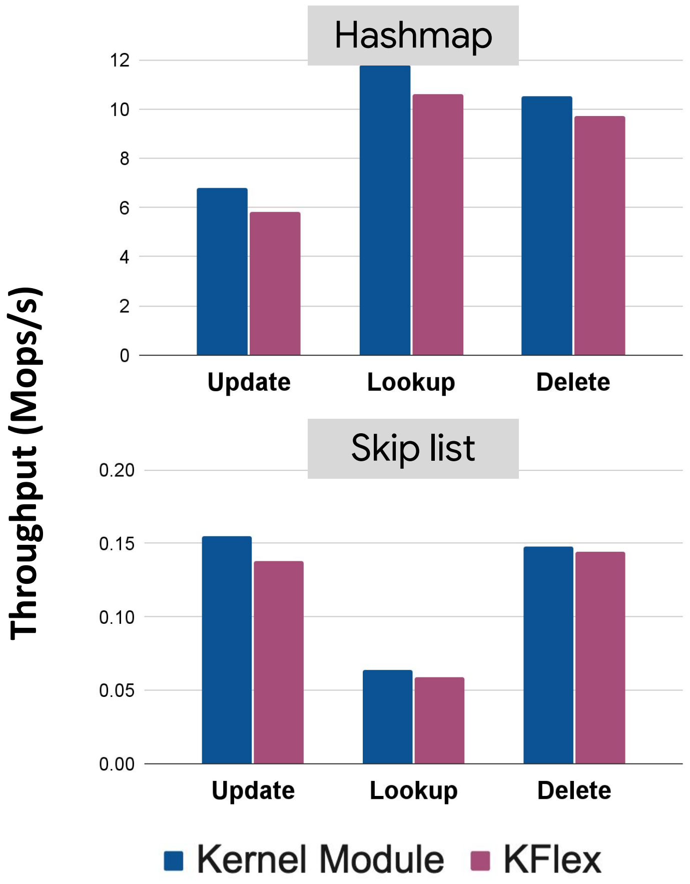
Low overhead

KFlex enables significant throughput improvements

Data Structures



Data Structures

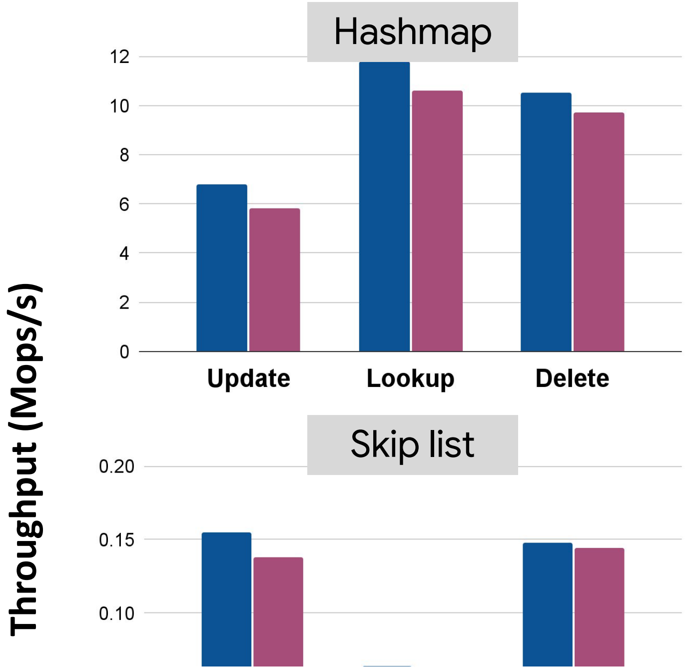


Offload arbitrary data structures

7% throughput overhead

30% latency overhead

Data Structures



Offload arbitrary data structures

7% throughput overhead

30% latency overhead

Implement infeasible functionality at low overhead

More results in the paper!

Latency numbers for Memcached

Throughput + latency numbers for Redis

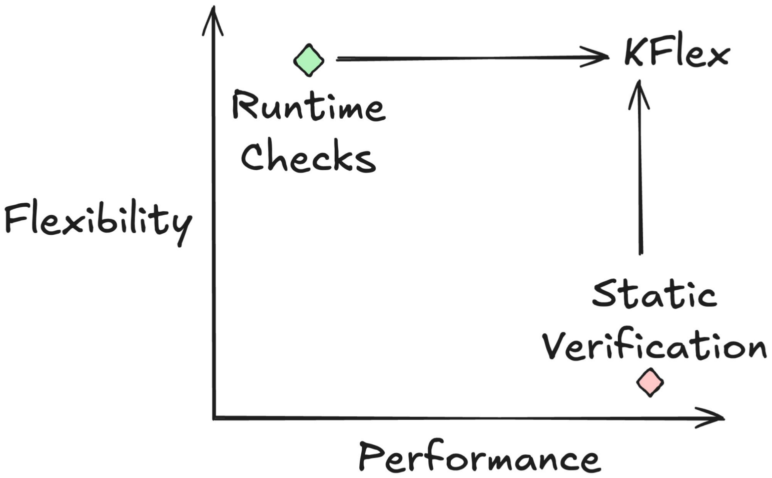
Impact of co-designing runtime mechanisms with verification

KFlex: fast, flexible, and practical kernel extensions

- Separate kernel safety into two sub-properties
  - Use distinct, bespoke mechanisms to enforce each sub-property
  - Co-design runtime mechanisms with verification to reduce overhead

KFlex: fast, flexible, and practical kernel extensions

- Separate kernel safety into two sub-properties
    - Use distinct, bespoke mechanisms to enforce each sub-property
    - Co-design runtime mechanisms with verification to reduce overhead
- Integrated into the upstream Linux kernel



Project website