Fast, Flexible, and Practical Kernel Extensions

Kumar Kartikeya Dwivedi† Rishabh Iyer* Sanidhya Kashyap†
†EPFL *UC Berkeley

Abstract

The ability to safely extend OS kernel functionality is a longstanding goal in OS design, with the widespread use of the eBPF framework in Linux and Windows demonstrating the benefits of such extensibility. However, existing solutions for kernel extensibility (including eBPF) are limited and constrain users either in the extent of functionality that they can offload to the kernel or the performance overheads incurred by their extensions.

We present KFlex: a new approach to kernel extensibility that strikes an improved balance between the expressivity and performance of kernel extensions. To do so, KFlex separates the safety of kernel-owned resources (*e.g.*, kernel memory) from the safety of extension-specific resources (*e.g.*, extension memory). This separation enables KFlex to use distinct, bespoke mechanisms to enforce each safety property—automated verification and lightweight runtime checks, respectively—which enables the offload of diverse functionality while incurring low runtime overheads.

We realize KFlex in the context of Linux. We demonstrate that KFlex enables users to offload functionality that cannot be offloaded today and provides significant end-to-end performance benefits for applications. Several of KFlex's proposed mechanisms have been upstreamed into the Linux kernel mainline, with efforts ongoing for full integration.

1 Introduction

OS kernel extensions enable applications to share the same core OS codebase, while specializing it based on their requirements by incorporating application-specific functionality into the kernel. After receiving significant attention in the 90s [23, 39, 66, 74], kernel extensions have been thrust into the spotlight once again with the introduction of the eBPF framework in Linux [41] and Windows [5]. eBPF-based kernel extensions are widely deployed today. For instance, Meta is known to load 50–150 eBPF extensions on each of their servers [27], and Google, Netflix, Cloudflare, Apple, Alibaba, and Dropbox have also documented several uses for eBPF in production, such as better observability, faster and more efficient networking, and improved security [17, 75].

Ideally, a framework for kernel extensibility must provide four key properties: *safety*, *flexibility*, *performance*, and *practicality*. Safety requires guaranteeing that extensions do not compromise the integrity of the kernel and other applications or extensions. Flexibility enables users to write extensions that offload diverse functionality to the kernel. Performance requires that extensions do not incur excessive runtime overheads when executed as part of the kernel. Finally, practicality dictates that the framework should be easy for developers to use and must not require them to learn new programming languages or use specific toolchains.

Existing solutions for kernel extensibility do not provide all four properties simultaneously and typically ensure safety along with only two of the three remaining properties. For instance, systems that use type- and memory-safe programming languages to guarantee safety [23] offer good performance and flexibility, but are not widely adopted because they require developers to learn new programming languages. Similarly, systems that rely on runtime checks to guarantee safety [66] offer flexibility and ease of use, but provide poor performance due to the large number of runtime checks required for safety. Finally, systems that rely on automated bytecode verification (*e.g.*, eBPF) have low runtime overhead and can support various programming languages. However, they sacrifice flexibility because automated verification techniques do not scale to arbitrary code [62, 81], which limits the functionality that users can offload.

We present KFlex—a new approach to kernel extensibility based on the observation that kernel safety comprises two distinct sub-properties, each best enforced by a distinct mechanism. KFlex divides kernel safety into (1) *kernel-interface compliance*, which requires extensions access *kernel-owned* resources (*i.e.*, kernel memory and kernel functions) only as permitted, and (2) *extension correctness*, which requires that extensions access *their own memory* safely and terminate correctly. To enforce kernel-interface compliance, KFlex uses automated bytecode verification since accesses to kernel-owned resources must satisfy *semantic* requirements beyond memory safety (*e.g.*, maintaining invariants on kernel data structures). In contrast, to enforce extension correctness, KFlex relies on lightweight runtime checks. This is because accesses to extension-owned resources need not satisfy semantic requirements, and simple runtime mechanisms are

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP '24, November 4–6, 2024, Austin, TX, USA
© 2024 Association for Computing Machinery.
ACM ISBN 979-8-4007-1251-7/24/11...$15.00
https://doi.org/10.1145/3694715.3695950

sufficient to guarantee that accesses to the extension's memory are within bounds, and ensure safe termination.

KFlex's use of distinct mechanisms to guarantee safety enables it to provide a better balance between flexibility, performance, and practicality. By using automated bytecode verification, KFlex eliminates the need for users to use specific languages and toolchains. By augmenting verification with lightweight runtime mechanisms that scale to arbitrary code, KFlex ensures that users can safely offload a wide range of functionality. Finally, the combined use of the above mechanisms ensures that KFlex incurs low runtime overhead since the guarantees provided by verification limit the required number of runtime checks.

We design and implement KFlex in the context of Linux—we reuse the eBPF framework for automated verification of kernel-interface compliance and augment it with a lightweight runtime that enforces extension correctness. KFlex retains the instruction set of eBPF's bytecode [7] and the interface through which eBPF extensions interact with the Linux kernel. This design choice makes KFlex less flexible than systems designed for extensibility [23] since Linux exposes a narrower interface to the eBPF framework. However, it ensures that KFlex is immediately useful to practitioners since Linux is the de-facto OS of the cloud, and KFlex can support existing eBPF extensions without modification.

The KFlex runtime relies on two key techniques, both *co-designed with eBPF's verifier* to ensure low performance overhead. First, KFlex leverages lightweight software fault isolation (SFI) [74] to sanitize all accesses performed by the extension for its own memory. KFlex's SFI uses eBPF's range analysis (§3.2) to elide checks when accesses for its own memory are verifiably safe, which significantly reduces the runtime overhead. Second, KFlex introduces *extension cancellations*: a mechanism that safely terminates long-running extensions at near-zero runtime overhead (§3.3). This low overhead stems from KFlex's use of the eBPF verifier to *statically* compute the set of kernel resources acquired by an extension. Several of KFlex's techniques have been upstreamed into the Linux kernel mainline [33, 34], with an ongoing effort for full integration [20].

We demonstrate that KFlex enables users to offload functionality that cannot be offloaded using eBPF today, which provides significant end-to-end performance benefits for applications. For example, we show how users can use KFlex to offload both `set()` and `get()` operations in Memcached. This flexibility enables the KFlex-based Memcached to outperform an eBPF-based Memcached [42]—which can only offload `get()` operations—and Memcached running in user space up to 2.83× and 3.01×, respectively. Similarly, we show how users can offload arbitrary data structures to the kernel, which enables the—currently unsupported—offload of Redis functionality and provides a 1.65× improvement in throughput compared to its user-space counterpart. Finally, we show how KFlex's flexibility enables users to co-design extensions with user-space applications, which is necessary for complex, production-grade applications.

In summary, this paper makes the following contributions:

- **Separation of safety properties.** We present a new approach to kernel extensibility that separates safety into two sub-properties and leverages bespoke mechanisms to enforce each property.
- **Runtime safety mechanisms.** We present runtime mechanisms co-designed with eBPF's automated verification to guarantee safe termination and memory safety at low overhead.
- **Practical design.** We demonstrate that our approach provides significant performance benefits for real applications and enables users to offload new functionality to Linux without having to use specific programming languages and toolchains.

KFlex is publicly available at https://rs3lab.github.io/KFlex.

2 Background and Motivation

In this section, we first define the target properties for any kernel extension framework (§2.1) before describing existing approaches and their shortcomings (§2.2).

We define kernel extensions as event handlers loaded by user-space applications into the kernel to process specific kernel events, such as system call invocations or packet arrivals. These extensions typically serve two purposes: modifying kernel functionality (*e.g.*, disallowing specific inputs to system calls), or offloading application-specific functionality to the kernel (*e.g.*, intercepting and processing packets belonging to a particular application). In this work, we treat both classes of extensions as one and hence use the terms offloads and extensions interchangeably hereafter.

2.1 Target Properties

Safety. To be safe, kernel extensions must satisfy two properties. First, they must conform to the kernel-provided interface, *i.e.*, they must access kernel-owned memory objects and invoke kernel functions only as permitted. We refer to this as *kernel-interface compliance*. Second, extensions must execute correctly, *i.e.*, they should access their own memory safely and provably terminate. We refer to this as *extension correctness*.

Flexibility. Users should be able to express diverse functionality in their extensions. Flexibility is determined by (1) the interface that the kernel exposes to extensions, and (2) the programming model available to developers for writing their extensions. The former is dictated by the design of the kernel, while the latter is usually a function of the mechanisms employed to ensure extension safety.

Performance. Extensions should incur (1) low runtime overhead when executing as part of the kernel and (2) low

communication overhead when interacting with applications in user space. Both aspects of performance are critical to modern applications. Applications today frequently offload their "fast path" to the kernel [42, 49, 78, 83, 84], thereby enabling the fast path to run at near-hardware I/O speeds and avoid the overhead of complex kernel I/O stacks. As I/O speeds increase rapidly [24] and processing times shrink to the microsecond scale [21], even small runtime overheads can significantly impact the performance of the application's fast path [22, 63]. Further, as applications are becoming increasingly complex, it is infeasible to offload them entirely to the kernel [42, 67]. As a result, low-overhead communication between extensions and user space is increasingly necessary.

Practicality. To facilitate adoption, users should not have to learn new programming languages or use specific toolchains to write kernel extensions. The kernel-extension framework should also be backward compatible and allow gradual adoption of new features without disrupting existing extensions.

2.2 Prior Approaches to Safe Kernel Extensibility

We now discuss the three most relevant prior approaches to safe kernel extensibility. Table 1 summarizes the pros and cons of each approach, which we gradually explain in the rest of this section. We describe other approaches in §7.

Safe languages. This approach—first proposed by the Spin OS [23] and later revisited by Singularity [45] and TockOS [51]—involves writing new OSes in type- and memory-safe languages, with the safety properties of the language guaranteeing that extensions execute safely. Since these OSes are designed with extensibility as a first-class goal, they provide maximum flexibility, both in terms of the interface that the OS exposes to extensions and the functionality that developers can express within extensions. However, adopting them in practice is challenging as they compel users to use new OSes, and learn and use specific programming languages and toolchains.

The emergence of Rust [14] as a memory-safe language for systems programming has sparked renewed interest [47, 51, 57, 58] in this approach, however, we do not know of a general-purpose kernel-extension framework that uses Rust. We chose not to use Rust since we wanted KFlex to generalize beyond a single programming language. We see the use of Rust as complementary to KFlex, and discuss this further in §6.

Software Fault Isolation (SFI). This approach—first proposed by VINO [66]—leverages runtime checks to sandbox the execution of extensions. The runtime checks typically include: (1) bounds checks on memory accesses to ensure memory safety and (2) a transactional mechanism for safely aborting extensions and rolling back possible side effects (such as acquired kernel object references).

This approach offers flexibility and ease of use since runtime checks can guarantee safety for arbitrary code written

Approach	Flexibility	Performance	Practicality
Safe languages (*e.g.*, SPIN [23])	✓	✓	×
Software Fault Isolation (*e.g.*, VINO [66])	✓	×	✓
Static verification (*e.g.*, eBPF [41])	×	✓	✓

Table 1. Summary of existing approaches to safe extensibility.

in diverse programming languages,. However, relying exclusively on runtime checks to guarantee safety typically leads to prohibitive performance overheads, which prevents these solutions from being widely used.

Static verification. This approach—first described by Necula *et al.* [61] and later adopted by the eBPF framework [41]—involves automatically verifying that extensions are safe before loading them into the kernel. This approach offers both performance and practicality. The use of verification eliminates runtime checks, and since the analysis is performed either on the binary or on bytecode, developers can use diverse programming languages.

Since we use eBPF as a starting point for KFlex, we provide additional background on how it works. eBPF is a register-based virtual machine in the Linux kernel. Users write extensions as event handlers for specific kernel events (*e.g.*, packet arrivals [44]). The extensions are then compiled down to eBPF bytecode [7], statically verified for safety by an in-kernel verifier, and finally loaded at the kernel "hook" for the designated kernel event.

eBPF extensions interact with the kernel through a *well-defined interface* which exposes kernel objects via hook-specific inputs or helper functions. This interface is currently more limited than the corresponding interface in OSes designed with extensibility as a primary goal [23] due to eBPF being retrofitted to Linux. However, this interface is gradually expanding with new extension hooks and helper functions being introduced in response to emerging use cases [43].

To keep the automated verification of kernel safety tractable, the eBPF framework enforces two constraints on extensions. First, since automated verification does not scale to arbitrary data structures [62, 81], eBPF prevents extensions from defining data structures and requires that they use a specific set of kernel-provided data structures (referred to as eBPF maps). Second, to guarantee termination, eBPF disallows loops that do not have statically computable loop bounds. For example, even a simple loop that iterates over a list (*e.g.*, `while (node->next != NULL)`) will be rejected by the eBPF verifier since termination cannot be statically guaranteed. This constraint also limits the use of synchronization primitives within an extension. For example, eBPF extensions today can only acquire a single lock because the verifier cannot guarantee deadlock avoidance when extensions acquire more than one lock instance.

While eBPF has been adopted in both Linux and Windows due to its performance and practicality, the above constraints limit its flexibility and are a frequent pain point for practitioners [42, 43, 64, 73, 84]. The current approach

to addressing this issue involves incrementally expanding the verifier's capabilities [30, 31, 35]. However, this strategy presents two major challenges. First, the process of merging new capabilities into the Linux kernel is time-consuming and labor-intensive, creating a substantial barrier for developers. Second, it leads to an increase in the complexity of the verifier, which increases the likelihood of bugs in kernel code. Recent work [47] uncovered several such bugs, highlighting the severity of this issue. These drawbacks demonstrate that the current case-by-case approach to addressing eBPF's limited flexibility is infeasible in the long term, and a more comprehensive and scalable solution is required.

Summary. Existing approaches to kernel extensibility fail to provide all four target properties simultaneously. Approaches based on new programming languages are rarely adopted in practice, runtime-based approaches incur prohibitive performance overheads, and verification-based approaches limit the functionality that users can offload. In the next section, we describe KFlex: a composite approach to kernel extensibility that provides an improved tradeoff between flexibility, performance, and practicality.

3 KFlex Design

KFlex's design is based on the observation that the two sub-properties that comprise safety—kernel-interface compliance and extension correctness—are best enforced by distinct mechanisms. Automated verification is best suited to enforce kernel-interface compliance since accesses to kernel-owned resources must satisfy semantic requirements beyond memory safety (e.g., maintaining invariants on kernel data structures). Further, since extensions can only access kernel-owned resources via well-defined interfaces (*e.g.*, hook-specific inputs and helper functions), automated verification techniques are sufficient to guarantee the safety of kernel-owned resources. In contrast, runtime checks are best suited to enforcing extension correctness since the kernel only requires extensions to be hang-free and memory-safe when accessing extension-owned memory. To keep the required runtime checks simple, KFlex allocates the extension-owned memory in a dedicated portion of the kernel's virtual address space, thus eliminating memory aliasing between kernel-owned and extension-owned memory.

We realize KFlex in the context of Linux using the eBPF framework as a starting point. We chose to do so since Linux is the de-facto OS of the cloud, and eBPF-based extensions are widely used. KFlex retains the instruction set of eBPF's bytecode and the interface through which extensions interact with the Linux kernel. This makes KFlex fully backward-compatible with eBPF and allows it to run existing eBPF-based extensions without modifications.

Figure 1 presents an overview of KFlex. The KFlex workflow involves three main steps: In step ❶, KFlex takes as input the extension as eBPF bytecode and uses the eBPF verifier to check kernel-interface compliance. Since KFlex retains the interface that Linux exposes to eBPF extensions, we reuse eBPF's kernel-interface compliance checker. In step ❷, KFlex's instrumentation engine (Kie) adds two types of instrumentation to the extension bytecode: (i) it sanitizes all heap accesses to prevent out-of-bounds access for the heap, and (ii) instruments loops to ensure that KFlex can terminate them safely. During this step, Kie leverages the eBPF verifier's range and loop termination analysis to reduce the emitted instrumentation and thus reduce the runtime overhead. The instrumented bytecode is then passed to the eBPF JIT, which compiles it down to machine code. KFlex augments the eBPF JIT to ensure that the added instrumentation is correctly compiled to native instruction sequences. Finally, in step ❸, the KFlex runtime executes the extension while guaranteeing memory safety and safe termination.

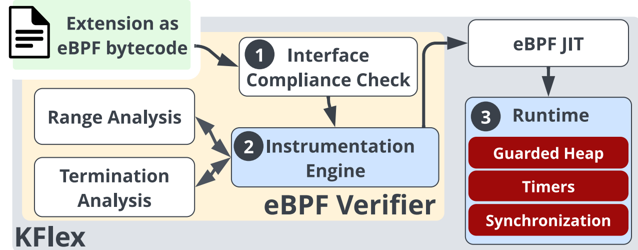

Figure 1. Overview of KFlex. Blue boxes denote components introduced by KFlex, and transparent boxes denote eBPF components that KFlex relies upon.

We now introduce KFlex's programming model (§3.1), before describing how it guarantees memory safety (§3.2) and safe termination (§3.3) at low runtime overheads. Finally, we discuss how KFlex enables extensions to transparently share memory with user-space applications (§3.4).

3.1 Programming Model

Like eBPF, KFlex extensions are written as event handlers that process specific kernel events. Listing 1 shows an example of a KFlex extension that processes incoming network packets at the XDP hook [44]. This extension implements a simple key-value store that manages a linked list of key-value pairs and can serve update and delete requests. The extension first parses the packet (line 17), then checks if the key is present in the linked list (lines 24–28), and if so, updates or deletes the value based on the request type (lines 37– 44).

KFlex provides APIs that enable developers to declare a memory region that is fully owned and managed by the extension; we call this region the *extension heap*. Developers declare extension heaps using the `kflex_heap(size)` macro, which specifies the size of the heap in GB (line 7). Once this heap is declared, developers can use `kflex_malloc()` and `kflex_free()` to allocate and de-allocate memory from the heap, create arbitrary memory layouts, and define their own data structures just like they would in user space (line 44). Table 2 summarizes the APIs that KFlex provides.

Type	API	Functionality
Macro	`kflex_heap(size)`	Heap declaration
Ext	`void *kflex_malloc(size_t)`	Allocate heap memory
	`void kflex_free(void *)`	Deallocate heap memory
	`void kflex_spin_lock(lock_t *)`	Acquire a lock
	`void kflex_spin_unlock(lock_t *)`	Release a lock

Table 2. Summary of APIs provided by the KFLEX runtime.

KFLEX also permits extensions to contain complex loops for which it is infeasible to statically compute loop bounds. For instance, our example key-value store iterates over a linked list using a possibly non-terminating loop condition (lines 24–28). KFLEX also provides a queue-based spin lock [56] that enables developers to synchronize access to shared memory between user-space applications and extensions. Unlike in eBPF, where extensions cannot acquire more than one lock at any point in time [28], KFLEX extensions can safely hold multiple instances of KFLEX's spin lock simultaneously.

KFLEX imposes one notable constraint on extensions. Specifically, KFLEX requires that loops in the extension code converge for kernel-owned resources, *i.e.*, any kernel resources acquired within a loop iteration (*e.g.*, line 33) must also be released by the end of that iteration (line 47). We believe this restriction—which we inherit due to use of eBPF's existing loop analysis logic [30]—is reasonable because most extensions do not acquire kernel-owned resources monotonically. We did not require such a loop when offloading Memcached, Redis, and five data structures using KFLEX.

3.2 Memory Safety Using Lightweight SFI

When an extension declares a heap, the KFLEX runtime maps the heap (aligned to its size) into the extension's virtual address space. The KFLEX runtime initializes the KFLEX memory allocator with per-CPU lists for each size class and a global list for memory conservation akin to how user-space allocators are initialized today. KFLEX does not pre-allocate physical memory for the entire heap at initialization, but instead populates the page table entries corresponding to the heap upon demand from the KFLEX memory allocator.

KIE implements SFI, which "guards" [74] all pointer dereferences in the extension code and ensures they access memory within heap bounds. This guarding mechanism is similar to other SFI implementations [74, 79] and involves two steps. First, KFLEX masks the pointer's address, keeping only the bits needed to offset into the heap correctly. Next, KFLEX calculates a sanitized address by adding the heap's starting address as a base. This ensures the sanitized address is safe, as the heap is aligned to its size.

As an example of how KIE's guarding mechanism works, consider a heap of size 256 bytes mapped between addresses 256 and 511 in the extension's virtual address space, and an unsafe pointer pointing to address 524. The masking step will zero out all but the lower 8 bits (since $2^8 = 256$), resulting in the value of 12. This masked address is then added to the

Listing 1 Example KFLEX extension, implementing a key-value store backed by a linked list.

```c
1  struct elem { // Linked list structure
2      int key;
3      int value;
4      struct elem *next, *prev;
5  };
6
7  kflex_heap(16); // 16 GB heap allocation
8  struct elem *head; // Linked list head
9  kflex_spinlock_t lock; // Lock used to protect the linked list
10
11 SEC("xdp")
12 int prog(struct xdp_md *ctx) {
13     struct bpf_sock_tuple tup;
14     struct bpf_sock *sk;
15     int proto, key;
16
17     if (!check_ipv4_udp(ctx)) // Check XDP packet type
18         return XDP_DROP;
19     // Initialize tuple for lookup from the XDP packet
20     init_sock_tuple(ctx, &tup);
21
22     // Extract the key of the request from the XDP packet
23     key = get_key(ctx);
24     kflex_spin_lock(&lock);
25
26     struct elem *e = head;
27     while (e != NULL) {
28       if (e->key != key) {
29         e = e->next;
30         continue;
31       }
32       // Only handle packets for existing UDP sockets
33       sk = bpf_sk_lookup_udp(ctx, &tup, sizeof(tup.ipv4), 0, 0);
34       if (!sk)
35         break;
36       // Extract the type of request from the XDP packet
37       switch (get_request_type(ctx)) {
38       case 0: // Update value for the element
39         // Extract value from XDP packet, and store it
40         e->value = get_value(ctx);
41         break;
42       case 1: // Delete element from linked list, and free it
43         list_delete(head, e);
44         kflex_free(e);
45         break;
46       }
47       bpf_sk_release(sk);
48       break;
49     }
50
51     kflex_spin_unlock(&lock);
52     return XDP_DROP;
53 }
```

heap's base address (256), leading to safe access at memory address 268. Note that if the pointer was originally within the heap, this sanitization process does not change the address.

We chose to sanitize heap accesses using address masking, as opposed to trapping and canceling extensions upon an out-of-bounds access since the former is known to provide better performance [54, 72]. This is because address masking can typically be optimized down to one hardware instruction (§4.2) while trapping and canceling requires extra hardware instructions for each access sanitization. Since kernel safety does not require any semantic guarantees on accesses to extension-owned memory, we chose the SFI scheme that provides better performance in KFLEX.

KFlex further reduces the overhead of SFI by leveraging the eBPF verifier's taint and range analysis to elide guards when heap accesses are provably safe. Leveraging the verifier's range analysis is particularly useful as it can ensure safety in cases where pointers are manipulated by adding or multiplying scalar values. Such scalar pointer manipulations are common in systems code, especially when accessing different fields within a `struct`.

Finally, KFlex's SFI also supports a "performance mode" that allows practitioners to trade off confidentiality for improved performance. In performance mode, KFlex does not sanitize read instructions, which reduces overhead but leads to a loss of confidentiality because extensions can read arbitrary kernel memory. Performance mode does not impact safety guarantees since writes are always sanitized, and reads that lead to page faults will trigger cancellations (§3.3).

We envision performance mode being used by practitioners who deploy extensions in a trusted environment. Examples of such scenarios include in-kernel tracing and observability contexts [12], as well as extensions deployed by a privileged user [68], which is the most common deployment scenario for eBPF-based extensions today.

3.3 Safe Termination Using Extension Cancellations

Enforcing safe termination requires aborting extensions after a pre-determined quantum and restoring the kernel to a quiescent state (*i.e.*, a state where invariants on kernel-owned memory and objects are satisfied) so that the system can make forward progress.

KFlex introduces extension cancellations: a mechanism that terminates long-running extensions and safely releases any kernel-owned objects acquired by the extension (*e.g.*, reference counters, locks) upon termination, to comply with kernel invariants. When combined with automated verification of kernel-interface compliance—which guarantees that accesses to kernel-owned memory comply with kernel invariants—-extension cancellations ensure that KFlex restores the kernel to a quiescent state when an extension's execution is terminated. Note that KFlex only guarantees that the kernel is restored to a quiescent state from which it can make forward progress. It does not do the same for forcibly terminated (and thus incorrect) extensions.

For each extension, KFlex defines a set of *cancellation points (Cps)*, which represent the instructions in the extension code at which it will terminate the extension if required.

Cps consists of two classes. Class 1 ($C1$) consists of all back edges of loops in the extension code for which termination cannot be guaranteed statically (line 49 in Listing 1). These represent scenarios where extensions might run indefinitely and hence might need to be forcibly terminated. Class 2 ($C2$) consists of accesses to the extension heap (lines 28, 29, and 40). Canceling the extension at all such accesses may be necessary because, although the SFI ensures that heap accesses are within bounds, it does not guarantee that the page being accessed has a valid physical address since that would require pre-allocating physical memory for the entire heap. Note that all Cps are defined in the extension code itself. Extensions are not terminated while executing trusted kernel helper functions. This ensures that KFlex does not have to rollback partial execution of kernel code upon a termination.

Under the covers, KFlex reduces the two classes of Cps into one by adding a heap access to the back edges in $C1$. For example, KFlex instruments the back edge of a potentially non-terminating loop, such as `while (node.next != NULL)`, by inserting a heap access (`*terminate`) in its body before the back edge, where `terminate` is a valid heap address. Spin locks implemented in extensions with a potentially non-terminating loop (*e.g.* in case of deadlocks) will be treated similarly. The accessed heap location contains a valid address for all back edges initially.

To ensure that the runtime can safely terminate the extension at Cps, Kie computes a unique *object table* for each Cp. This object table records the stack locations and registers holding kernel-owned objects acquired by the extension when the Cp is executed and thus represents the resources that must be safely released in case the extension is forcibly terminated at the Cp. In Listing 1, Cps at lines 28 and 29 will have no object tables as no kernel resources are held at those points. However, the Cp at line 40 will have an object table with an entry for the socket object acquired at line 33 with its corresponding destructor (`bpf_sk_release`).

KFlex relies on the eBPF verifier's symbolic execution to compute object tables. Since an extension can only acquire and release kernel resources via helper functions with well-defined semantics, KFlex can precisely track the set of resources held by the extension at each Cp, as well as the destructor required to release these resources upon a forcible termination. Since KFlex requires all loop iterations in extension code to release any kernel resources acquired during the same iteration, the object table for each Cp is also independent of the number of loop iterations executed and thus can be uniquely determined. Note, there still exist possible corner cases in the computation of the object table due to different sequences of non-loop branches leading to the same Cp. We describe how KFlex addresses these cases in (§4.3). Kie passes the computed tables to the KFlex runtime along with the instrumented bytecode.

To enforce cancellations for long-running loops, the KFlex runtime monitors how long an extension has been executing for and zeroes out the address (`terminate`) used for the heap access added to loops when the execution time exceeds the desired quantum. This results in the extension raising an exception at the Cp corresponding to the long-running loop. The KFlex runtime catches this exception and then releases all kernel-owned objects acquired by the extension before aborting the program. To correctly release kernel-owned objects, the KFlex runtime steps through the unique object

table corresponding to the CP and calls the corresponding destructor for each object in the table. In scenarios where heap accesses to unmapped memory (C2 CPs) raise an exception, the runtime terminates the extension as above, *i.e.*, it catches the exception and releases all acquired kernel resources using the object table.

Thus, by statically analyzing the kernel resources acquired by an extension, KFLEX is able to guarantee safe termination at near-zero runtime overhead for correct extensions (*i.e.*, extensions that terminate on their own). Specifically, KFLEX's only runtime overhead for correct extensions comes from the additional heap access performed (`*terminate`) at every loop iteration. In our experience, this overhead is negligible since `*terminate` will typically be in the CPU's L1 cache due to its repeated use [46].

3.4 Low Overhead Communication with User Space

We now describe how KFLEX enables low-overhead communication between extensions and user-space applications. KFLEX provides such a communication channel since modern applications are growing increasingly complex and cannot always be entirely offloaded using kernel extensions [42, 67].

KFLEX enables user-space code to request extension heaps be mapped into their address space, thus providing direct and transparent access to all extension state through virtual memory and eschewing system calls.

Providing such direct access to extension state requires overcoming two key challenges. First, merely mapping the heap into the extension and user address space is not sufficient, as all references to memory must point to the right addresses. For example, consider a linked list that both the extension and user-space application walk by accessing a pointer to the linked list structure residing in the shared heap. When this pointer is communicated by the extension to the application, it will be a pointer in the extension's address space, and will likely lead to a page fault when accessed in user space.

The second challenge involves ensuring the safe sharing of mutable state between user-space applications and extensions. While both user space and extensions can use synchronization mechanisms, such synchronization is made more complex since extensions running in the kernel can preempt their user-space counterparts at any point. For example, an extension can preempt its user-space counterpart in the middle of a critical section and attempt to acquire the same lock, leading to a deadlock. Although extension cancellations ensure that such extensions will eventually be terminated, such deadlocks will likely lead to significantly worse application performance.

To address the first challenge, KIE translates pointers to the shared heap into a valid user-space address whenever they are *stored*. This translation is similar to sanitization, but the base address is adjusted to point to the location where the shared heap is mapped in the application's virtual address space. This translation does not affect extension correctness because the address is checked the next time it is dereferenced in the extension. KFLEX allows developers to disable the "translate on store" feature. Disabling it reduces the runtime overhead of the extension, which can be beneficial for performance-critical paths. However, developers will need to modify their applications to handle the translation of stored accesses in user space. While KFLEX supports this alternative, in this work, we focused on running unmodified user-space applications and hence perform the translation on stores in the extension code.

To address the second challenge, KFLEX enables user-space applications to request a temporary time slice extension, similar to Symunix [38], when holding a spin lock that an extension might also acquire. This approach works well in practice because spin lock critical sections are typically short, and thus a single time slice extension—which we set to 50µs—usually suffices for the application to complete the critical section. Once the additional time slice expires, the application is forcefully preempted, ensuring forward progress of the system.

In scenarios where a non-cooperative user-space process does not release the lock and is forcefully preempted, extensions in the kernel waiting on the lock will initially continue to spin waiting for it. However, these extensions will eventually be canceled by KFLEX's extension cancellation mechanism. When such a cancellation occurs, the KFLEX runtime unloads the extension from the kernel but does not destroy the extension heap, as it may be used for backing allocations in the user-space application. The extension heap is de-allocated only when the application closes the heap file descriptor explicitly or when the application terminates.

4 Implementation

We implement KFLEX as part of the Linux kernel v6.9. In total, we implement KFLEX in 15k LoC, with 2.5k LoC for the changes to the eBPF verifier and JIT, 8k LOC for implementing the KFLEX runtime, and a further 4.5k LoC for unit tests. Our code passes all the tests in the eBPF test suite, ensuring backward compatibility and no regressions for existing extensions.

4.1 Extension Heaps

KFLEX implements extension heaps as eBPF maps to allow applications to reference and `mmap()` the heap using a file descriptor object. KFLEX creates heaps using the `bpf(2)` system call [2], and the kernel allocates the heap in the `vmalloc` [10] region with an alignment request equal to the size of the heap. The KFLEX runtime then maps this heap in user space, before it loads the extension that will utilize this heap. This allows the extension's verification and JIT procedure to take into account the base address of the user-space mapping, which is used to concretize the address into the extension's

JITed code for pointer translation. The physical memory allocated for a heap is tracked as part of the application's memory `cgroup` [3]. This accounting ensures that any resource limits on the application also apply to the memory allocated by its kernel extensions.

When allocating an extension heap, KFlex allocates two guard pages on either side of the heap to ensure safety. We choose the guard page size to be 32 KB (2^{15}) because eBPF load and store instructions allow the addition of a signed 16-bit offset to a pointer loaded from a register [6]. These offsets can range from $-2^{15}+1$ to 2^{15}. The guard pages ensure that any memory access performed by the extension remains within the memory mapped to the extension's address space.

Unfortunately, while such guard pages are necessary, they cause fragmentation in kernel memory since KFlex's SFI requires heaps to be mapped to a location aligned to their size for address translation to work. For example, assuming two extensions each request a heap of size 4 GB, the kernel cannot map them into a contiguous region because the guard pages of the first will force the kernel to skip the following 4 GB chunk to maintain alignment requirements. We describe possible solutions to this fragmentation in §6.

Memory allocator. We implement the KFlex's memory allocator backend in user space on top of `jemalloc` [8]. Objects for the extension heaps are allocated using extent hooks API for `jemalloc arenas` [1], where page allocations are served from the extension heap mapped in user space instead of `mmap()`. These are then forwarded to the per-CPU cache of objects maintained for extensions to allocate memory (§3.2). A user-space thread spawned by the KFlex runtime monitors and refills the per-CPU caches when they run low.

As a demonstration of KFlex's flexibility, we implement the `kflex_malloc()` and `kflex_free()` functions as KFlex extensions. These extensions, which share data structures with user space using the mechanisms in §3.4, enable the KFlex memory allocator to respond to allocation/deallocation requests quickly, particularly if the allocation/deallocation can be served from the extension's cache.

4.2 The KFlex SFI

The KFlex SFI efficiently implements address sanitization for `x86_64` as follows. To extract the offset bits from a heap address, KFlex emits a single `AND` instruction. All accesses (loads, stores, atomic RMWs) performed on the heap address then use `x86`'s indexed addressing mode to access memory. To avoid having to load the heap address mask and the heap base address into `x86` hardware register for each heap access, KFlex reserves the unused `x86` registers `R12` and `R9` (for the eBPF ISA on the `x86` target in Linux) to hold the base address and mask respectively. Since `R12` is a callee-saved register, we modify the eBPF JIT to push it upon entry into the kernel extension, and since `R9` is caller-saved, we ensure that the JIT reloads it with the mask each time a kernel or extension function is invoked.

Performance mode. To ensure that un-instrumented reads in performance mode cannot be used to read user-space data—that is controlled by possibly malicious applications—and steer the control flow of kernel extensions, KFlex relies on the fact that extensions run with `SMAP` [76] enabled. As a result, extensions will trap on accessing any user-space address, leading to cancellation.

4.3 Extension Cancellations

Object table generation. To ensure that KFlex computes unique object tables per Cp, we have to account for the case where different types of kernel resources are present in the same stack location or registers for a given Cp. Such a scenario occurs when the program arrives at the same Cp via a different sequence of non-loop branches, such that an entry in the object table cannot accurately describe the disjunction of distinct kernel resources. We mitigate such cases by spilling conflicting kernel resources upon acquisition to unique stack locations in the frame. For all the extensions we wrote for our evaluation and all the extensions in the eBPF test suite in the kernel, we did not encounter such code generation from the compiler. Nevertheless, KFlex still addresses this corner case.

Monitoring execution duration. The KFlex runtime implements passive watchdog-driven monitoring of extensions at the granularity of seconds. We implement this by using Linux's softlockup and hardlockup watchdogs [16], which enable us to keep track of stalls in interruptible and non-interruptible eBPF extensions, respectively. For sleepable eBPF extensions [29], the KFlex runtime spawns a background task that periodically wakes up and checks stalled extensions that are blocking tasks in a sleepable context.

Returning control to the kernel. Upon completing stack unwinding, KFlex returns a default error code to the kernel. KFlex chooses the default value based on the kernel hook where the extension is loaded; for instance, security extensions must deny by default, and network extensions should pass packets by default. KFlex also provides users with the flexibility to attach a callback function to their extension that modifies the returned error code to suit their needs. Naturally, this callback function is restricted in terms of allowed behavior; for instance, it cannot contain other cancellation points (to avoid recursive cancellations), and cannot contain unbounded loops.

Cancellation scope. Since the `*terminate` heap access is shared by all invocations of an extension across CPUs, cancellation on one CPU due to non-termination also leads to termination of the same extension on other CPUs. We made this policy decision to avoid running buggy extensions

inside the kernel. As future work, cancellations due to non-termination can be scoped to individual invocations of an extension on a given CPU, without affecting other CPUs.

4.4 Low Overhead Communication with User Space

KFlex implements time slice extensions by introducing a counter in the `rseq` region [26, 32] and incrementing and decrementing it on lock acquisition and release. Such a design ensures that nested locks are correctly accounted for. In cases where the user-space task does not finish executing the critical section in the allotted extension, KFlex forcefully preempts it, which results in extensions waiting for the lock, eventually stalling, and being canceled. We consider this acceptable since we believe that KFlex's only safety concern is to ensure that the kernel's integrity is maintained and that it can continue making forward progress. Repairing the execution of faulty applications is out of scope for KFlex.

4.5 Integrating KFlex into the Linux Kernel

Thus far, we have upstreamed several building blocks for KFlex, such as initial support for cancellations in the verifier [34], and associated stack unwinding logic [33]. We have also proposed upstream patches implementing object table generation for cancellations [20], and performance mode [36], which are under review. In parallel to our work, Alexei Starovoitov contributed eBPF arenas [19] upstream, which achieves feature parity with KFlex's heaps, but supports a maximum of 4 GB size due to a different SFI scheme. We are actively migrating our extensions to arenas, and will pursue integration of our SFI scheme and optimizations used for KFlex heaps upstream to allocate arenas bigger than 4 GB in size.

5 Evaluation

We evaluate KFlex by answering the following questions:

- Does KFlex deliver end-to-end performance benefits for real-world applications? (§5.1)
- Does KFlex provide tangible flexibility benefits by enabling developers to offload new functionality to the kernel? (§5.2)
- Does KFlex enable developers to co-design extensions with user-space applications? (§5.3)
- Does KFlex's co-design of runtime techniques with eBPF's verifier provide meaningful reductions in runtime overhead? (§5.4)

Testbed: We use a testbed set up as per RFC 2544 [13] with two directly connected machines: a server that runs KFlex or the baselines and a client that runs a closed-loop load generator. Both machines are identical, with a 96-core (192-thread) Intel Xeon Platinum 8468 CPU running at 2.30GHz, 512GB of RAM, and an Intel X710 10 Gbps NIC. Both machines run Ubuntu 24.04, with the server machine running our modified version of Linux v6.9.

In all our experiments, the client machine runs a closed-loop load generator with 64 threads and 16 clients per thread, while the server runs KFlex or the baselines using 8 threads. All measurements are performed at the client, ensuring end-to-end evaluation of KFlex. Unless otherwise specified, we use a key size of 32B and a value size of 64B for experiments involving key-value lookups. The clients generate requests according to a Zipfian access pattern with $s = 0.99$. Each experiment runs for 30 seconds, and we discard the first 10% of samples to remove warm-up effects. We ran our experiments for several minutes and obtained similar results.

5.1 Performance Benefits for Applications

A key use-case for kernel extensions today is the potential performance benefits for the fast path of networked applications whose service times are approaching the microsecond-scale [22, 63]. Hence, to evaluate KFlex's performance benefits, we use it to offload Memcached and Redis, two widely deployed, microsecond-scale key-value stores.

Memcached. To offload Memcached with KFlex, we implement all the Memcached logic that parses incoming network packets and processes GET and SET requests in a single KFlex extension. This ensures that the KFlex-based Memcached runs entirely in kernel space with no user-space involvement. We attach the KFlex extension to the XDP hook [44] (*i.e.*, the Linux hook to process incoming ethernet packets). Since SET requests in Memcached run over TCP, we implement support in Linux to handle TCP's fast path at the XDP hook itself. Our KFlex-based Memcached consists of approximately 2000 LoC.

We compare the performance of the KFlex-based Memcached to two baselines: Memcached running in user space (referred to as Memcached hereafter) and BMC [42]. BMC is a recent research proposal that implements a look-aside cache to handle only GET requests with eBPF. BMC preallocates memory in the kernel for its look-aside cache. BMC does not offload SETs since processing SET requests requires dynamic memory allocations, which vanilla eBPF does not provide. In contrast, KFlex enables the offloading of both SETs and GETs since developers can use the KFlex memory allocator to allocate memory on demand.

We use three workloads to evaluate the performance of KFlex-Memcached, each with a different ratio of GETs:SETs (90:10, 50:50 and 10:90, respectively). Put together, these workloads enable us to evaluate KFlex's performance benefits in read-heavy, mixed, and write-heavy scenarios, all of which are common in production today [70, 77]. For each workload, we set key and value sizes to be 32B each; we had to reduce the value size since BMC does not support values larger than their keys. For each workload, we measure the throughput achieved by each system, along with the 99th percentile latency.

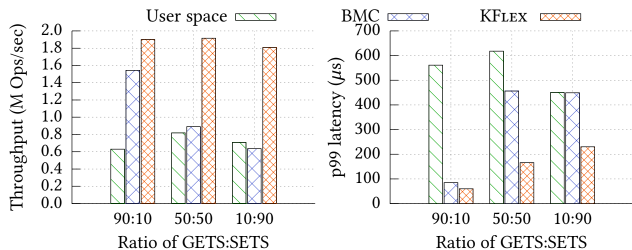

Figure 2. Comparison of Memcached (8 threads) performance when offloaded using KFlex, eBPF and when run in user space.

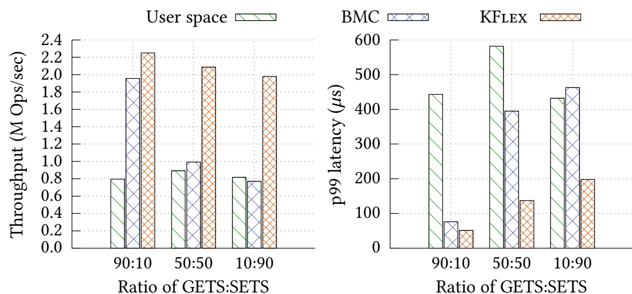

Figure 3. Comparison of Memcached (16 threads) performance when offloaded using KFlex, eBPF and when run in user space.

Figure 2 (a) illustrates KFlex's throughput benefits for all three workloads. We observe that KFlex-Memcached sustains 1.23×-2.83× and 2.33×-3.01× higher throughput than BMC and Memcached, respectively. KFlex's improvements over BMC only increase as the fraction of SETs increases because BMC offloads only GET requests. Meanwhile, KFlex's improvements over Memcached remain similar since the key difference is that the former does not pay the overhead of the Linux TCP stack or the cost of context switch.

Figure 2 (b) illustrates the 99th percentile latency for each workload for all three systems. We observe similar trends as for throughput with KFlex-Memcached providing a 1.41×-1.95× and 1.95×-9.35× lower tail latency than BMC and Memcached, respectively.

To demonstrate that KFlex's performance benefits hold irrespective of the number of threads used to run the application, we repeat the above experiments with 16 threads for each Memcached instance. Figure 3 depicts the results and demonstrates that KFlex's performance benefits are similar despite the change in the number of threads.

Redis. To offload Redis, we implement the logic necessary to parse requests and process SET, GET, and ZADD requests in a single KFlex extension akin to Memcached. Since all of the above request types run over TCP, we attach the extension to the `sk_skb` hook (*i.e.*, the Linux hook to process packets once they have been processed by the transport layer). Our implementation of Redis totals approximately 3100 LoC.

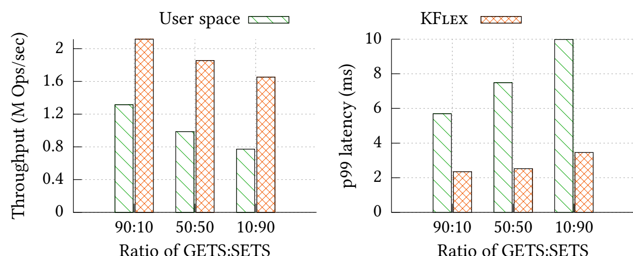

Figure 4. Comparison of Redis performance when offloaded using KFlex and when run in user space.

While user-space Redis is a single-threaded, KFlex-Redis can run on multiple threads in the kernel. So, to keep the comparison fair, we use a parallel version of Redis (KeyDB [9]) as our user-space baseline. We retain the workloads and metrics we used for Memcached to evaluate KFlex-Redis.

Figure 4 illustrates KFlex's performance benefits for all three workloads. We observe that KFlex-Redis sustains 1.61×-2.14× greater throughput than KeyDB while providing 0.97×−2.96× lower tail latency.

While KFlex's performance benefits are significant (ignoring the outlier), they are lower for Redis in comparison to Memcached. This difference is due to the corresponding extensions being attached to different hooks. Since Redis runs SETs and GETs over TCP, while Memcached runs GETs over UDP and SETs over TCP [42], all requests in KFlex-Redis must traverse the Linux TCP stack before being processed by the extension, resulting in reduced performance benefits.

Takeaway. Based on the above results, we conclude that KFlex provides significant throughput and latency benefits for low latency, microsecond-scale applications when compared to running these applications in user space or offloading them using eBPF.

5.2 Offloading New Functionality with KFlex

Recall that one of eBPF's major limitations today is the lack of support for extension-defined data structures. We now demonstrate that KFlex overcomes this limitation and enables developers to define complex data structures in their extensions and offload functionality that critically depends on such data structures.

Offloading new data structures. We use KFlex to implement a hash table, linked list, red-black tree, skiplist, and two network sketches. Each data structure is defined entirely within an extension and does not rely on the user space application, even for initialization. All data structures except the hash table are single-threaded, but adding support for concurrent operations only requires further engineering.

We evaluate the performance of these data structures by measuring the throughput and average latency for lookup, delete, and update operations. We evaluate the throughput and latency with and without performance mode enabled (§4.2). As a baseline, we use an identical implementation

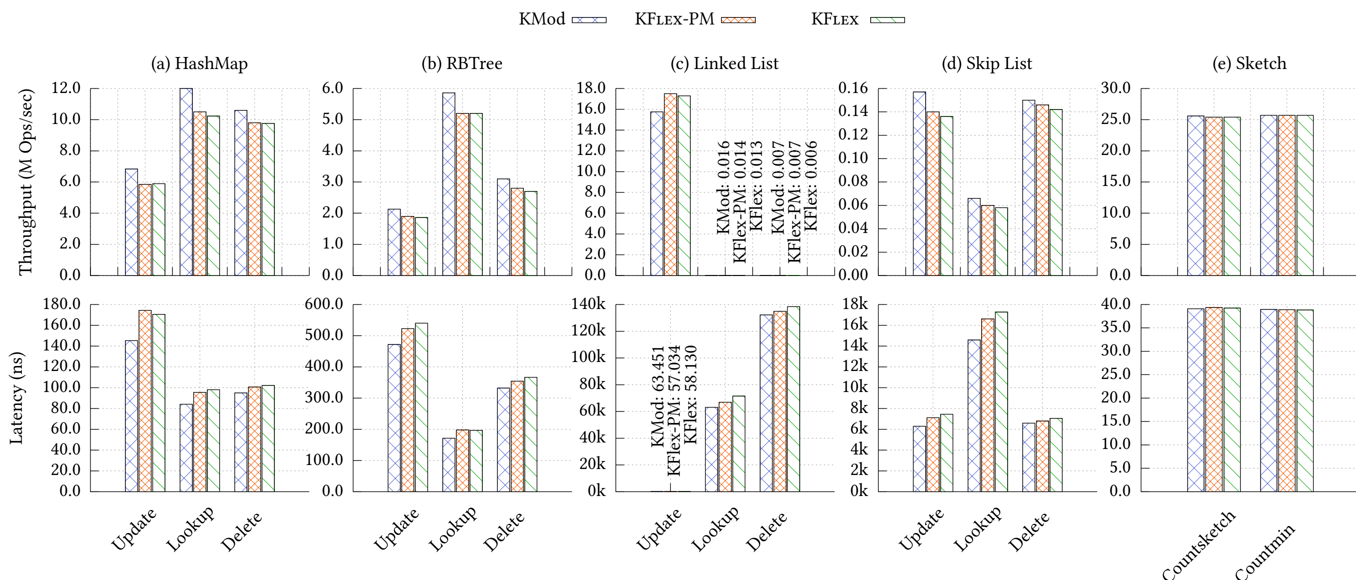

Figure 5. Comparison of single-threaded update, lookup, and delete operations' performance for five data structures implemented in user space and offloaded to the kernel using KFlex. Linked list update is a constant time operation while lookup and delete traverse the list of 64K elements.

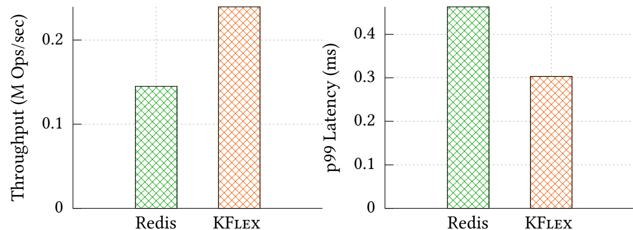

Figure 6. Comparison of ZADD performance of Redis with the KFlex-offloaded version that uses the skiplist data structure.

written as a kernel module (i.e., unsafe kernel code). This baseline represents the maximum achievable performance since it incurs zero runtime overheads. For uniformity, we run each data structure in a single thread.

Figure 5 illustrates the performance of the KFlex-based data structure offloads. We see that, on average, the KFlex-offloaded data structures suffer a throughput overhead of 9% throughput and a latency overhead of 31.7% when performance mode is disabled. When performance mode is enabled, we see that the latency overhead reduces by 3-4% for pointer-chasing heavy data structures (linked list, skip list), 1-2% on average for other data structures (hashmap, rbtree), and no change for network sketches. Note, the KFlex-based implementations also pay the cost of inefficiencies in the eBPF ISA, such as register pressure and poor implementations of `memcpy()`. We believe this overhead can be reduced with additional engineering, but (as we will show next) it may not significantly impact end-to-end application performance.

Offloading ZADD processing in Redis. We now demonstrate the value of KFlex's added flexibility for *application developers* by offloading the processing of ZADD requests to our KFlex-offloaded Redis. The ZADD operation adds one or more keys to a sorted set or updates the corresponding score if the key already exists. Redis implements ZADD using a combination of a hashmap and a skiplist. It uses the hashmap to manage keys associated with requests, with the value in the hashmap pointing to a skiplist. Note that ZADD poses a significant challenge to KFlex's flexibility since it requires new data structures (i.e., a skiplist) to be allocated in the fast path (whenever a new key is added to the hashmap)

Offloading ZADD using eBPF today is infeasible since eBPF does not provide a skiplist implementation. In contrast, KFlex provides the flexibility to offload ZADD while retaining the above implementation. This is because developers can not only define, iterate through and update a skiplist from within the extension but also allocate it on demand using KFlex's memory allocator.

We evaluate the performance of KFlex's implementation of ZADD for Redis by comparing its performance with Redis running in user space. We use the same setup to evaluate performance but use only a single thread on the server since Redis' ZADD implementation acquires a global lock on the hashmap for each operation.

Figure 6 illustrates KFlex's performance benefits for both throughput and 99th percentile latency. We observe that KFlex outperforms the user-space version by 1.65× and reduces the tail latency by 52.8%. These results demonstrate that while implementing data structures using KFlex does incur a small runtime cost, this cost pales in comparison to the end-to-end performance benefits that KFlex's extensibility provides.

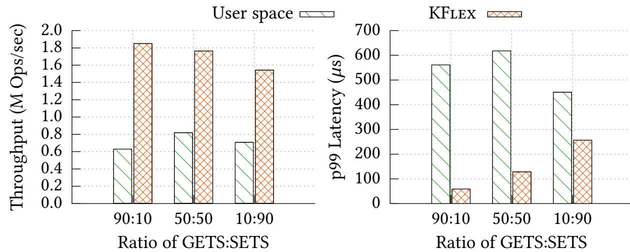

Figure 7. Comparison of Memcached performance when co-designed with KFlex versus running in user space.

Takeaway. Based on the above results, we conclude that KFlex overcomes the most significant limitation of eBPF today and allows developers to define, iterate through, update, and even allocate data structures on demand from within a kernel extension.

5.3 Co-designing Extensions with User-Space Applications

We now demonstrate how developers can use KFlex to co-design extensions with code running in user space. Such co-design enables application threads running in user space and extensions to work in tandem to handle different types of scenarios. For instance, a common design philosophy is to execute auxiliary functions that perform some non-critical tasks sporadically (*e.g.*, garbage collection, lazy resource accounting and statistics generation). These tasks are often ignored in benchmarks but are necessary when running applications in production.

As an example of co-design, we describe the implementation of garbage collection for Memcached. Garbage collection is a background operation that involves removing objects from the cache either when its occupancy has crossed a certain threshold or when an object has expired. By default, Memcached performs garbage collection every 1s. Given this infrequent operation, offloading garbage collection to the kernel is sub-optimal since it will take crucial CPU time away from applications, given its higher privilege level.

KFlex allows developers to run garbage collection in user space while offloading Memcached's fast path to the kernel using shared pointers (§3.4). These pointers let garbage collection threads access Memcached's hash table, which is defined in the extension's heap. Without shared pointers, Memcached would need to run entirely in user space to support garbage collection.

We use KFlex to implement a version of Memcached that performs garbage collection as follows: Garbage collection is performed using user-space threads that wake up every 1s, while the fast path logic is identical to the implementation we used to evaluate Memcached performance for GETs and SETs. We use spin locks to synchronize accesses to shared state. We could not rely on mutexes since the fast path, which runs in the kernel, cannot sleep.

Function	Total number of guard insns.	Guards elided
Linked list update	4	4 (100%)
Linked list lookup	1	1 (100%)
Linked list delete	2	2 (100%)
Hashmap update	10	8 (80%)
Hashmap lookup	2	0 (0%)
Hashmap delete	4	3 (75%)
RBTree update	16	16 (100%)
RBTree lookup	2	2 (100%)
RBTree delete	31	27 (87%)
Skiplist update	15	10 (66%)
Skiplist lookup	3	2 (66%)
Skiplist delete	9	4 (44%)

Table 3. Reduction in guard instructions emitted by the KFlex SFI due to guarantees provided by the eBPF verifier's range analysis. We do not show numbers for the two network sketches since the safety of all memory accesses in the sketch can be verified statically.

We compare the performance of our co-designed Memcached with Memcached running in user space using the same workloads we used in §5.1.

Figure 7 presents KFlex's performance benefits for both throughput and 99th percentile latency. We observe that KFlex continues to outperform the user-space version and provides 2.2–2.9× improvement in throughput and reduces the tail latency by 42.8%–89.5% reduction in tail latency across the three workloads.

These results are as expected. The improvement in throughput drops slightly (was 2.33×-3.01× without garbage collection) since the fast path now occasionally contends with the slow path when accessing the same hash table. While this contention occurs in user-space Memcached too, the impact is less significant since its overall service time is much longer. Similarly, the improvement in tail latency is also reduced (it was 1.95×-9.35× without garbage collection). This reduction is more significant, but it is understandable since contention over the shared hash table leads to similar latency spikes in both user space and the kernel.

Takeaway. Based on the above results, we conclude that KFlex's shared pointers and translation mechanism enables users to co-design applications with their extensions, and reap the performance benefits of kernel offloads even in production scenarios where auxiliary functionality such as garbage collection and logging are necessary.

5.4 Does Verification Reduce SFI Overhead?

Finally, we demonstrate the benefits of co-designing KFlex's SFI with eBPF's static verifier. To do so, we measured the number of SFI guard instructions elided due to the verifier's range analysis for the update, lookup, and delete operations on each data structure. We do not include guard instructions emitted on forming a new heap pointer in the total, as those must not be optimized away. Since each guard instruction is identical, the number of guard instructions elided serves as a proxy for the reduction in performance overhead.

Table 3 details the results. We observe that the verifier's range analysis elides 76% of the guard instructions emitted on pointer manipulation on average across all data structures and elides 100% for several data structure operations as well. The guard instructions that the range analysis cannot elide are typically those for which the size of the scalar added to the pointer may be larger than the heap size. Nevertheless, we see that co-designing KFlex's SFI with the verifier is crucial for achieving low runtime overhead.

Evaluation summary. In our evaluation, we demonstrate that KFlex not only provides significant end-to-end performance benefits for applications with stringent performance requirements (*e.g.*, Memcached and Redis) but also provides tangible flexibility benefits by enabling developers to define and use their own data structures in extension code, and thus offload functionality that cannot be offloaded to Linux today. We then demonstrate the extent of KFlex's flexibility by showing how it enables developers to co-design extensions with user-space applications and thus accommodate auxiliary functionality necessary in production while retaining the performance benefits of kernel offloads. Finally, we show how co-designing KFlex's runtime mechanisms is crucial to its low runtime overhead and overall performance benefits.

6 Discussion

What we would like to change about eBPF.

Register pressure. The low register count in the eBPF ISA [7] limits the number of native hardware registers available for use by extensions after JIT. This leads to greater register pressure for code written in extensions. While the performance downsides are less pronounced on x86_64, they can be significant for other architectures with no shortage of registers, such as ARM64.

SIMD instructions. Currently, eBPF does not provide access to vector instructions. This does not make a difference in the context of the Linux kernel (as SIMD instructions are disallowed in kernel mode) but could lead to a significant performance overhead on other OSes. That said, the eBPF ISA can be extended with support for vector instructions and FPU state save/restore can be done around extension execution to allow their use.

eBPF memory model. eBPF currently lacks a formal memory model, which hurts the portability and maintenance of synchronization primitives written as extensions. In KFlex, we follow the Linux Kernel Memory Model [69] at source level, and rely on implementation details of the `x86` eBPF JIT for correct code translation. We plan on following the existing effort to introduce an eBPF memory model [55], and adopting it upon standardization.

KFlex's flexibility limitations due to Linux's interface. In comparison to OSes designed for extensibility from the ground up (*e.g.*, SPIN [23], Singularity [45]) Linux exposes a much more restricted interface to eBPF-based extensions. This restricted interface is understandable since eBPF was retrofitted to the Linux kernel. However, it does have flexibility limitations; for example, one of SPIN's canonical examples was being able to offload a web server into the kernel; this is currently infeasible using eBPF (or KFlex) since Linux does not permit extensions to read files or perform blocking I/O.

That said, we believe that these limitations are not fundamental. First, due to eBPF's growing popularity, the Linux kernel is incrementally widening the interface it exposes to eBPF extensions [37, 43, 50]. Second, while the design of KFlex in this paper is Linux centric, we think the key insight of combining static bytecode verification with runtime mechanisms applies broadly.

Possible alternate designs for KFlex.

Stacking safety properties. While this work addresses the safety of the kernel where extensions are deployed, it does not address functional correctness and memory safety of extensions themselves (*e.g.* in their own heap). We believe our design's clear separation of concerns in terms of its responsibilities (*i.e.* kernel safety) allows for a more flexible approach of achieving stronger safety and functional correctness for extensions. For instance, KFlex extensions written in Rust (as supported by eBPF) would have stronger intra-heap memory safety guarantees than the ones written in C. Such an approach allows composition of diverse toolchains and frameworks to stack various safety properties for an extension's functional behavior on top of kernel safety guarantees provided by KFlex. As a future direction, we will explore integrating verification toolchains, such as Dafny [4], with eBPF's LLVM backend to compose provably correct kernel extensions.

Faster extension stall recovery. The watchdog-driven cancellation approach used by KFlex (§4.3) operates at second-granularity to detect non-terminating extensions, which may not be ideal for some users. Similar mechanisms are used in existing virtual machine runtimes (*e.g.*, WebAssembly), such as runtime metering [11], and epoch-based interruption. We chose against runtime metering, as it requires pervasive instrumentation of the extension to account for instructions executed at runtime, and would cause prohibitive runtime overhead. Epoch-based or timer-based interruption is not always applicable for kernel extensions, which run in interrupt-disabled contexts. Thus, as future work, we will explore alternative mechanisms such as time sampling using hardware clocks in extensions at loop back edge Cps to expedite recovery time and operate at sub-second time scales.

Scaling heap regions. Alignment constraints on the virtual addresses of heaps limit the number of maximum heap regions (§4.1). We plan to explore heap domain striping [60], where Intel MPK is used to mark adjacent heap domains

with distinct protection keys and eliminate guard pages to perform dense packing of regions with the same size.

7 Related work

VINO Transactions. VINO provides support for aborting grafts (or extensions) using a transactional runtime mechanism. At first glance, these transactional semantics suggest stronger safety guarantees compared to KFlex, specifically three of the four ACID properties: atomicity, consistency, and isolation [66]. However, the interpretation of these properties in VINO diverges from their conventional definitions in database systems. In VINO, extensions interact with and modify kernel data through accessors, but these accessors do not stage their mutations. Instead, all side effects take immediate effect and are visible throughout the system before the transaction is committed. As a result, atomicity and isolation apply only between aborts and individual accessor calls. Consistency in this context ensures that aborting after an accessor call does not violate kernel invariants. Accessors that require explicit actions to restore the kernel state log their changes in an 'undo log,' ensuring that all entries are processed upon transaction abort. Once a transaction commits, the 'undo log' is discarded. This mechanism is used to release acquired kernel resources and locks upon aborts. Therefore, as detailed in §3.3, KFlex's extension cancellations provide equivalent safety guarantees to VINO's transactions.

Safe Kernel Device Drivers. Many existing systems have proposed fault isolation techniques for kernel extensions in the context of device drivers. Nooks [71] achieves isolation for device drivers through hardware page protection mechanisms, but incurs high overhead. BGI [25] combines static analysis with runtime checks, but is not tailored to the narrow interface of kernel extensions. Thus, it incurs performance overhead due to an increase in runtime checks for memory and type safety, since extension memory is not separated from kernel memory. SafeDrive [85] is similar, but provides weaker isolation and temporal safety than BGI.

Moving Kernel Functionality into User Space. An alternative approach to extension safety is to run extensions in user space to restrict their permissions. This approach, which minimal kernels such as the Exokernel [39] and microkernel-based systems [18, 52] implement, has the advantage of ensuring flexibility, since user-space applications need not be constrained to specific programming models. We believe such approaches are particularly suited to platforms such as embedded and mobile devices where minimal kernels are primarily used [15, 51]. However, since the cloud relies almost exclusively on monolithic OSes today, we design for KFlex for such OSes.

Software Fault Isolation. Our approach of SFI is inspired by the seminal work of Wahbe *et al.* [74], which used guard sequences utilizing bitwise math and use of reserved hardware registers to optimize code instrumentation. Since then, subsequent research work has made improvements to the implementation, instrumentation, and mechanisms used for SFI enforcement, such as PittSFIeld [53, 54] and XFI [40]. Projects such as Google's NaCl [80] have demonstrated the use of segmentation to sandbox arbitrary code, but are limited by the sandbox size due to use of segmentation registers. KFlex takes inspiration from SFI systems that integrate verification with runtime checks [48, 59, 82], and co-design both together to improve correctness and reduce overhead.

Heaps for BPF. There have been related efforts at approximating heap abstractions for eBPF extensions. Barret Rhoden constructed makeshift heaps out of BPF array maps [65] and used bounds checking to construct data structures. Alexei Starovoitov implemented eBPF arena [19], which has a 4 GB bound on size, and utilizes pointer manipulation using 32-bit operations as its SFI scheme.

8 Conclusion

In this paper, we presented KFlex, a safe kernel extension framework that strikes an improved balance between flexibility, performance and practicality. KFlex achieves this balance by separating how the two properties that comprise safety are enforced. Specifically, it relies on static bytecode verification to guarantee that extensions conform to the kernel's interface, and runtime mechanisms co-designed with verification to ensure that they access their own memory safely and provably terminate.

We design and implement KFlex in the context of the Linux kernel and build on top of the eBPF framework. KFlex's key technical contributions include runtime techniques that are co-designed with eBPF's static verification to ensure particularly low overhead for extensions. Our evaluation demonstrates that KFlex provides significant performance benefits for real applications and enables users to offload new functionality to Linux without having to learn and use specific programming languages and toolchains. Several of KFlex's proposed mechanisms have been upstreamed into the Linux kernel mainline with an ongoing effort for full integration.

9 Acknowledgments

We thank our shepherd Andrew Quinn, as well as the anonymous reviewers for their feedback that greatly improved the paper. We also thank Barret Rhoden, Paul Chaignon, Shravan Narayan, Aurojit Panda, and James Larus for their feedback on drafts of the paper at various stages. Finally, we thank Alexei Starovoitov, Eduard Zingerman, and other eBPF maintainers for their feedback on our patches submitted to the Linux kernel. This work is supported by the SNSF project grant 212884 and, in part, by the eBPF Foundation.

References

[1] Jemalloc Arena Extent Hooks. https://jemalloc.net/jemalloc.3.html#arena.i.extent_hooks.

[2] bpf(2) — Linux Manual Page. https://man7.org/linux/man-pages/man2/bpf.2.html.

[3] CGroup v2. https://docs.kernel.org/admin-guide/cgroup-v2.html.

[4] The Dafny Programming and Verification Language. https://dafny.org/dafny/.

[5] Making eBPF work on Windows. hhttps://cloudblogs.microsoft.com/opensource/2021/05/10/making-ebpf-work-on-windows/.

[6] eBPF Instruction Manual. https://docs.kernel.org/bpf/standardization/instruction-set.html#basic-instruction-encoding.

[7] eBPF Instruction Set Specification, v1.0. https://docs.kernel.org/bpf/standardization/instruction-set.html.

[8] Jemalloc. https://jemalloc.net.

[9] KeyDB. https://docs.keydb.dev/.

[10] Linux Virtual Memory Map for x86_64. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt.

[11] Metering in WASM. https://ewasm.readthedocs.io/en/mkdocs/metering/.

[12] Linux Observability. https://www.oreilly.com/library/view/linux-observability-with/9781492050193/ch04.html.

[13] Benchmarking Methodology for Networking Interconnect Devices. https://www.ietf.org/rfc/rfc2544.txt.

[14] The Rust Programming Language. https://www.rust-lang.org/.

[15] seL4. https://en.wikipedia.org/wiki/L4_microkernel_family#High_assurance:_seL4.

[16] Softlockup detector and hardlockup detector. https://docs.kernel.org/admin-guide/lockup-watchdogs.html.

[17] The State of eBPF, 2024. https://www.linuxfoundation.org/hubfs/eBPF/The_State_of_eBPF.pdf.

[18] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. 1986.

[19] D. Alden. A proposal for shared memory in BPF programs, 2024. https://lwn.net/Articles/961941.

[20] D. Alden. Cleaning up after BPF exceptions, 2024. https://lwn.net/Articles/969185/.

[21] L. A. Barroso, M. Marty, D. A. Patterson, and P. Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 2017.

[22] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[23] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. J. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, 1995.

[24] Q. Cai, M. Vuppalapati, J. Hwang, C. Kozyrakis, and R. Agarwal. Towards μs tail latency and terabit ethernet: disaggregating the host network stack. In *ACM SIGCOMM Conference*, 2022.

[25] Castro, Miguel and Costa, Manuel and Martin, Jean-Philippe and Peinado, Marcus and Akritidis, Periklis and Donnelly, Austin and Barham, Paul and Black, Richard. Fast Byte-Granularity Software Fault Isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 45–58, 2009.

[26] J. Corbet. Restartable Sequences. https://lwn.net/Articles/650333/, 2015.

[27] J. Corbet. BPF at Facebook (and beyond). https://lwn.net/Articles/801871/, 2019.

[28] J. Corbet. Concurrency management in BPF, 2019. https://lwn.net/Articles/779120/.

[29] J. Corbet. Sleepable BPF programs, 2020. https://lwn.net/Articles/825415/.

[30] J. Corbet. Generic iterators for BPF, 2023. https://lwn.net/Articles/926041/.

[31] J. Corbet. Red-black trees for BPF programs, 2023. https://lwn.net/Articles/924128/.

[32] J. Corbet. User-space spinlocks with help from rseq(). https://lwn.net/Articles/944895/, 2023.

[33] J. Corbet. Stack unwinding with exceptions in eBPF, 2023. https://lwn.net/Articles/938435/.

[34] K. K. Dwivedi. Exceptions in eBPF - Linux Plumbers Conference, 2023. https://lpc.events/event/17/contributions/1578/attachments/1240/2521/Exceptions%20in%20BPF.pdf.

[35] K. K. Dwivedi. User-defined objects in eBPF, 2022. https://lore.kernel.org/bpf/20221118015614.2013203-1-memxor@gmail.com/.

[36] K. K. Dwivedi. Zero overhead PROBE_MEM, 2024. https://lore.kernel.org/bpf/20240619092216.1780946-1-memxor@gmail.com/.

[37] J. Edge. The FUSE BPF Filesystem, 2023. https://lwn.net/Articles/937433/.

[38] J. Edler. *Process Management for Highly Parallel Unix Systems*. Forgotten Books, 2016.

[39] D. R. Engler, M. F. Kaashoek, and J. W. O. Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, 1995.

[40] Erlingsson, Ulfar and Abadi, Martín and Vrable, Michael and Budiu, Mihai and Necula, George C. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88, 2006.

[41] M. Fleming. A thorough introduction to eBPF, 2017. https://lwn.net/Articles/740157/.

[42] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *18th USENIX Symposium on Networked Systems Design and Implementation*, 2021.

[43] T. Heo. BPF Extensible Scheduler Class, 2022. https://lore.kernel.org/bpf/20221130082313.3241517-1-tj@kernel.org.

[44] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, pages 54–66, 2018.

[45] G. C. Hunt and J. R. Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, 2007.

[46] R. R. Iyer, M. Unal, M. Kogias, and G. Candea. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

[47] J. Jia, R. Sahu, A. Oswald, D. Williams, M. V. Le, and T. Xu. Kernel extension verification is untenable. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, 2023.

[48] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan. SFI safety for native-compiled Wasm. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, Feb. 2021.

[49] M. Kogias, R. Iyer, and E. Bugnion. Bypassing the Load Balancer Without Regrets. In *ACM Symposium on Cloud Computing*, 2020.

[50] M. Lau. Overview of the BPF networking hooks and user experience in Meta - Linux Plumbers Conference 2022. https://lpc.events/event/16/contributions/1363/.

[51] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[52] J. Liedtke. On micro-kernel construction. *ACM SIGOPS Operating Systems Review*, 1995.

[53] S. McCamant and G. Morrisett. Efficient, Verifiable Binary Sandboxing for a CISC Architecture. 2005.

[54] S. McCamant and G. Morrisett. Evaluating SFI for a CISC Architecture. In *USENIX Security Symposium*, volume 10, pages 209–224, 2006.

[55] P. E. McKenney. BPF Memory Model. https://datatracker.ietf.org/meeting/118/materials/slides-118-bpf-bpf-memory-model-00.

[56] Mellor-Crummey, John M and Scott, Michael L. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[57] S. Miller, K. Zhang, M. Chen, R. Jennings, A. Chen, D. Zhuo, and T. E. Anderson. High Velocity Kernel File Systems with Bento. In *19th USENIX Conference on File and Storage Technologies*, 2021.

[58] S. Miller, A. Kumar, T. Vakharia, T. Anderson, A. Chen, and D. Zhuo. Agile Development of Linux Schedulers with Ekiben. In *Proceedings of the Ninteenth EuroSys Conference*, 2024.

[59] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: better, faster, stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 395–404, 2012.

[60] Narayan, Shravan and Garfinkel, Tal. Segue & ColorGuard: Optimizing SFI Performance and Scalability on Modern x86. In *The 17th Workshop on Programming Languages and Analysis for Security*, 2022.

[61] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. *SIGOPS Operating Systems Review*, 30(SI):229–243, 1996.

[62] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[63] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. E. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[64] A. Protopopov. BPF static keys, wildcard map, XXH3 Hash - LSF/MM/BPF, 2023. http://vger.kernel.org/bpfconf2023_material/anton-protopopov-lsf-mm-bpf-2023.pdf.

[65] B. Rhoden. eBPF Shenanigans with Flux - Linux Plumbers Conference, 2023. https://lpc.events/event/17/contributions/1601/.

[66] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, 1996.

[67] F. Shahinfar, S. Miano, G. Siracusano, R. Bifulco, A. Panda, and G. Antichi. Automatic Kernel Offload Using BPF. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, 2023.

[68] A. Starovoitov. Introduce CAP_BPF. https://lwn.net/Articles/820560/, 2020.

[69] A. Stern. Linux Kernel Memory Model. https://github.com/torvalds/linux/blob/master/tools/memory-model/Documentation/explanation.txt.

[70] M. Sutherland, B. Falsafi, and A. Daglis. Cooperative Concurrency Control for Write-Intensive Key-Value Workloads. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023.

[71] Swift, Michael M and Annamalai, Muthukaruppan and Bershad, Brian N and Levy, Henry M. Recovering device drivers. *ACM Transactions on Computer Systems (TOCS)*, 24(4):333–360, 2006.

[72] Tan, Gang. Principles and Implementation Techniques of Software-Based Fault Isolation. 1(3):19–20, 2017.

[73] H. Tao. Ternary Search Tree Proposal - BPF, 2022. https://lore.kernel.org/bpf/20220331122822.14283-1-houtao1@huawei.com/.

[74] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993.

[75] Wikipedia. eBPF's Adoption in Production, 2024. https://en.wikipedia.org/wiki/EBPF.

[76] Wikipedia. Supervisor Mode Access Prevention. https://en.wikipedia.org/wiki/Supervisor_Mode_Access_Prevention, 2024.

[77] J. Yang, Y. Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[78] R. Yang and M. Kogias. HEELS: A Host-Enabled eBPF-Based Load Balancing Scheme. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions, eBPF 2023, New York, NY, USA, 10 September 2023*, 2023.

[79] Z. Yedidia. Lightweight Fault Isolation: Practical, Efficient, and Secure Software Sandboxing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2024.

[80] Yee, Bennet and Sehr, David and Dardyk, Gregory and Chen, J Bradley and Muth, Robert and Ormandy, Tavis and Okasaka, Shiki and Narula, Neha and Fullagar, Nicholas. Native client: A Sandbox for Portable, Untrusted x86 Native Code, 2010.

[81] A. Zaostrovnykh, S. Pirelli, R. R. Iyer, M. Rizzo, L. Pedrosa, K. J. Argyraki, and G. Candea. Verifying Software Network Functions with No Verification Expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[82] L. Zhao, G. Li, B. De Sutter, and J. Regehr. Armor: Fully verified software fault isolation. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 289–298, 2011.

[83] Y. Zhou, Z. Wang, S. Dharanipragada, and M. Yu. Electrode: Accelerating Distributed Protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation*, 2023.

[84] Y. Zhou, X. Xiang, M. Kiley, S. Dharanipragada, and M. Yu. DINT: Fast In-Kernel Distributed Transactions with eBPF. In *21st USENIX Symposium on Networked Systems Design and Implementation*, 2024.

[85] Zhou, Feng and Condit, Jeremy and Anderson, Zachary and Bagrak, Ilya and Ennals, Rob and Harren, Matthew and Necula, George and Brewer, Eric. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 45–60, 2006.