kSFS: Repurposing a Microkernel-like Interface for Fast and Secure In-Kernel Linux File Systems

Dinglan Peng
Purdue University

Pedro Fonseca
Purdue University

Abstract

File systems are widely-used and crucial but notoriously complex and a major source of vulnerabilities in operating systems. Recent works have proposed introducing in-kernel sandboxing techniques to isolate kernel components including file systems. However, a well-defined and secure boundary, where all interactions between untrusted and trusted kernel components should be validated against a strong threat model, is often ignored. This lack of secure boundary particularly applies to Linux file systems, which rely on a large and complex interface and interact with many kernel subsystems such as VFS and block devices. Defining such an interface is a challenging prerequisite of sandboxed kernel file systems.

We address this challenge with kSFS, a framework for in-kernel sandboxed file systems. kSFS repurposes the FUSE protocol, which is a microkernel-like interface originally designed for user-space file systems in Linux, as a secure interface for untrusted sandboxed kernel file systems that has strong isolation guarantees. Furthermore, kSFS generalizes WebAssembly to kernel space as a generic sandboxing mechanism and achieves compatibility with existing user-space file system implementations with minimal porting effort. For instance, porting the NTFS and exFAT implementations from user space with kSFS required modifying fewer than 300 LoC. While achieving better security and reliability than Linux file system implementations, kSFS achieves significantly better performance than their user-space counterparts. For the real-world applications tar and RocksDB, the kSFS NTFS implementation achieves up to 29% and 60× better performance than the user-space baseline, respectively, and only 0% to 52% lower performance than the insecure Linux implementation.

1 Introduction

File systems are a major source of vulnerabilities in operating systems due to their large code bases, complex semantics, and large feature sets [12, 46]. For instance, since its introduction in 2008, Linux's ext4 file system implementation has been found to have more than 70 CVEs, which has the highest density of memory or concurrency bugs among all Linux subsystems [42]. This is especially concerning in monolithic kernels such as Linux, where a vulnerability in a single file system implementation may compromise the whole system.

Recent works have proposed applying in-kernel sandboxing and compartmentalization to operating system kernels to achieve isolation between kernel components. Linux has introduced eBPF for secure kernel extensions [66, 67], including file system-related ones such as ExtFUSE [8], which optimizes certain types of FUSE file systems. In addition to software-based sandboxing, another line of work uses hardware features, such as virtualization extensions and pointer authentication, to compartmentalize kernel components [49, 56].

In addition to isolated in-kernel execution, a well-defined isolation boundary with validation of the interactions between components assuming a strong threat model is equally important to strong security, despite being often ignored. The difficulty in finding a secure interface between components applies particularly to the standard Linux file systems, which rely on a notoriously large and complex interface and interact with many kernel subsystems, including VFS, page cache, and block devices. Moreover, this interface is designed to trust file system implementations and assumes they follow the correct semantics without isolation guarantees. Naively designing a new interface for a stronger threat model would be particularly difficult and error-prone, given that file systems are intermingled with the rest of the kernel, make frequent use of pointers that cross components, and generally have complex semantics and myriad corner cases.

Despite the monolithic architecture of Linux, we observe that the Linux kernel already provides a well-defined and secure file system interface – the FUSE protocol, which is one of the few microkernel-like interfaces in Linux. The FUSE protocol allows untrusted user-space file systems implementations to communicate with the kernel, validating all interactions with a strong threat model. For example, a user-space file system is prevented from corrupting or leaking sensitive core kernel data under malicious inputs. These guarantees are similar to what we expect for an isolated, in-kernel file system. In this paper, we explore the idea of *repurposing* microkernel-like interfaces as secure interfaces for isolated kernel components to partially achieve microkernels' [43] strong security guarantees without sacrificing efficiency through secure in-kernel execution. We validate our idea on a single but particularly important kernel component, the file systems, by proposing kSFS, a system that repurposes the FUSE protocol as a secure interface for sandboxed kernel file systems.

Several challenges need to be resolved to implement kSFS. The first challenge is to implement an in-kernel isolation mechanism that works for file systems, which usually have highly complex and large code bases. eBPF, the commonly used framework for in-kernel sandboxing, does not meet our requirements. First, it has very limited expressiveness. For example, eBPF programs are limited in length and cannot have custom data structures or even unbounded loops without sacrificing certain key guarantees [21]. Second, eBPF forces an unconventional event-driven programming model and requires using expensive *maps* for storing any global state. This introduces significant rewriting effort and runtime overhead [44] to existing code bases, especially stateful programs such as most file system implementations. Thus, to avoid reinventing the wheel by turning eBPF into a generic sandboxed language, we consider other alternatives that work better for our scenario. WebAssembly, a de facto standard for software-based fault isolation, has been well studied and deployed in security-critical settings – this includes sandboxed execution in web pages [73], browsers [53], and server-side applications [10, 11, 62], making it a good candidate for many use cases. Inspired by the success of WebAssembly, we introduce it into the Linux kernel as the isolation mechanism used by kSFS.

However, securely executing WebAssembly code in the kernel requires that we address another challenge: enforcing that sandboxed code must only access authorized memory and devices without crashing the system, hanging the kernel, or corrupting the core kernel state. kSFS addresses this challenge by generalizing the WAMR [10] WebAssembly runtime for the Linux kernel. Our experience shows that enforcing these guarantees in kernel space requires significant runtime changes, since the original runtime assumes user-space execution only. For example, the runtime handles signals for catching out-of-bound memory accesses, but the kernel does not provide similar interfaces for handling kernel page faults. Also, the runtime does not restrict WebAssembly code from occupying excessive CPU cycles, so a simple infinite loop can lead to denial-of-service attacks. We significantly modified the runtime's design and implementation to adapt to the kernel interface and enforce the isolation guarantees.

Another major challenge is to provide a compatible interface for existing file system implementations that minimizes porting effort. To address this challenge, in addition to the FUSE protocol, kSFS adds support for the WASI [63], which is WebAssembly's standard interface for accessing system resources such as files, and provides a POSIX-compatible API built upon the WASI interface. Moreover, kSFS implements a modified libfuse library that provides a C API for file system implementations to communicate with kSFS's kernel components using the FUSE protocol. Existing file systems using the interface of libfuse can be ported to kSFS by compiling them to WebAssembly code with the library linked. Conveniently, these file systems can still use the familiar POSIX API, e.g., `open()`, `read()`, etc., to access disks, despite running in the kernel. By providing the two APIs, kSFS allows easy adoption with few code modifications.

The FUSE protocol provides an interface with strong isolation. However, it is not designed for high performance: the FUSE protocol relies on a system call interface that causes numerous mode and context switches as well as unnecessary data copies. For example, our experiment in §7.4 shows that for RocksDB, the user-space implementation of NTFS using the FUSE interface results in 98% lower random read throughput than the Linux implementation. This causes another challenge: sandboxing file systems must not introduce significant performance overhead. To address this challenge, we design and implement several optimizations to the FUSE protocol. First, kSFS replaces the file-based interface of FUSE with in-kernel WebAssembly function calls to eliminate the overhead from mode and context switches as well as serialization. Second, kSFS provides a zero-copy API and optimizes synchronization to allow concurrent operations. With these techniques, kSFS achieves significantly better performance than FUSE, despite the WebAssembly instrumentation costs.

We apply kSFS to two real-world file systems, exFAT and NTFS, by porting their existing and mature FUSE implementations to kSFS, both with fewer than 300 lines of code changed. Our experiments show that kSFS can run the file system implementations in a secure in-kernel isolated environment without introducing large overhead, especially when compared with their user-space counterparts with similarly strong reliability and security properties. For instance, for the exFAT file system, the random read throughput of the kSFS implementation under 64 I/O threads is 31× higher than the FUSE one and is only 23% lower than the Linux one, despite our strong isolation guarantees. Moreover, our evaluation shows that kSFS improves the performance of real-world applications over FUSE file system implementations. In particular, for RocksDB, kSFS achieves 61× throughput of FUSE in the random read benchmark, and is only 2.4% lower than the Linux implementation. kSFS's security and performance benefits prove that microkernel-like interfaces can be repurposed and integrated with software-based in-kernel isolation techniques to close the gap between traditional microkernels and monolithic kernels in efficiency and security.

In short, the paper makes the following contributions: (1) an approach to building fast and secure in-kernel file systems by repurposing the FUSE protocol and applying WebAssem-

Table 1: Number of CVEs found in Linux file system implementations, with their mainline adoption year.

File System	Year	kLoC	#CVEs
Btrfs	2009	152	94
ext4	2008	65	72
XFS	2003	200	32
NTFS	2021	33	20
Bcachefs	2024	104	5
exFAT	2020	8	3

bly sandboxing; (2) an implementation of the approach, kSFS, which supports real-world file systems with minimal porting effort; and (3) a comprehensive evaluation on the performance and security improvement of kSFS using real-world file systems.

2 Motivation

In this section, we discuss the benefits and limitations of the existing approaches to securing file systems and present our approach, which improves security and performance.

2.1 File System Bugs

Monolithic kernels like Linux entirely run in supervisor mode, which means that all drivers are included in the TCB, significantly impacting security and reliability, since a bug in any driver can cause the entire system to fail. For example, Linux has evolved into a huge project with more than 30 million lines of code, including the core kernel and all the drivers that are written by different developers.

File systems are a major source of serious bugs in operating systems [12, 46]. Tab. 1 lists the number of CVEs found in several popular Linux file systems. For example, despite having only 65 kLoC in the most recent version, the ext4 file system has 72 CVEs since it was mainlined, many of which represent serious vulnerabilities. Moreover, a prior survey shows that the ext4 file system has the highest density of memory or concurrency bugs in all Linux subsystems [42]. This shows that writing kernel file system implementations is highly error-prone and critical for security and reliability. Furthermore, unlike device drivers that are only enabled when certain hardware is present, file systems impact the kernel as long as a corresponding file system is mounted. Thus, it is important to secure file systems.

2.2 Existing Approaches to Securing File Systems

There have been several approaches proposed to improve the security and reliability of file system implementations.

User-Space File Systems. While most commodity operating systems follow the monolithic architecture where the entire operating system runs in supervisor mode, many of them also provide interfaces for extending OS functions in user space [4,50,68,70,72]. For example, FUSE [68] is a user-space file system interface first implemented in Linux. Originally designed to simplify file system development by avoiding kernel-space programming, FUSE isolates user-space file systems from the kernel, which avoids most of the attacks on the kernel and significantly improves system security and reliability. Nevertheless, the low performance of user-space file systems due to frequent mode and context switches greatly limits their deployment, despite its programming and security benefits. Our evaluation in §7.4 shows that for the real-world application RocksDB, the FUSE implementation of NTFS has 98% lower throughput than the Linux implementation on random reads. Thus, user-space file systems have limited applicability in real-world deployments with high performance demand.

Memory-Safe Languages. Applying memory-safe languages to operating system kernels has been a recent active topic of research and has shown great potential [19, 42, 59]. For instance, since version 6.1, Linux has officially introduced Rust as a secondary programming language in addition to C. Before that, there were attempts at incorporating memory-safe languages to harden the Linux kernel. For example, Bento [52] introduces a framework for implementing Linux kernel file systems using Rust. However, although promising, memory-safe languages, such as Rust, still have important limitations. First, fully eliminating kernel vulnerabilities is beyond their capability [42], as memory safety can only prevent a subset of bugs and specifically does not guarantee kernel liveness. For example, Rust-verified memory-safe code can still cause kernel stalling by an infinite loop or kernel panic by an overflowing recursive function or even simply an `unwrap()`, all of which are considered safe in Rust's semantics [74] but cause unrecoverable outcomes in kernel space [76]. Second, migrating the existing device drivers or file systems from C to memory-safe languages requires significant development effort and prevents the reuse of the existing code bases.

In-Kernel Isolation. Recent works have proposed introducing in-kernel isolation techniques, e.g., eBPF, for secure kernel extensions, including those for the storage subsystem. For example, XRP [86] applies eBPF hooks to NVMe drivers to perform some simple operations, such as tree lookup without traversing the kernel storage stack. ExtFUSE [8] uses eBPF to accelerate some simple but frequent operations of certain FUSE file system implementations. These isolation techniques provide mechanisms for secure in-kernel execution of untrusted code. However, defining a secure interface between the untrusted code and the remaining kernel, such as the VFS subsystem and block devices, is critical to implementing sandboxed kernel file systems but is often ignored. Linux only provides a large and intrusive interface for kernel modules, including file systems, by exporting thousands of symbols and using complicated data structures, such as linked lists or nested structures, that are shared between components, making the validation of all interactions exceptionally difficult. A prior work, kSplit [32], applies static analysis to the interface used by a specific kernel driver to identify the shared state, such as specific structure fields, between the kernel and the driver for memory isolation. However, simply identifying the shared state involved and limiting file systems to only accessing it is insufficient for sandboxing, as the kernel interface is designed to trust file systems and does not validate the shared state.

2.3 Repurposing Microkernel-like Interfaces

The challenge of writing kernel file systems that are both secure and efficient and the limitations of existing approaches drive us to explore a new approach. As a monolithic kernel, Linux lacks well-defined and secure interfaces that validate the interactions between kernel components against a strong threat model. This lack of a secured boundary makes it difficult to compartmentalize Linux. Despite adopting a monolithic architecture for the bulk of the kernel, Linux also provides several microkernel-like interfaces, such as FUSE [68], UIO [70], and VFIO [72], which are designed for untrusted components. For example, the FUSE protocol is used to implement user-space file systems. In microkernels, most operating system components, like file systems, run in user space and communicate with the kernel via system calls instead of in-kernel function invocations. Moreover, all inputs from these components are untrusted and are validated by the kernel. This strong threat model from microkernels also applies to several Linux components, like the FUSE driver. We observe that these microkernel-like interfaces exactly satisfy the missing piece of compartmentalization – well-defined secure interfaces that validate untrusted inputs. Along with in-kernel isolation mechanisms, these microkernel-like interfaces can be repurposed for kernel compartmentalization to achieve similar isolation guarantees to microkernels as well as high efficiency like monolithic kernels.

Based on this observation, in this paper, we propose kSFS, a framework for sandboxed file system implementations in the Linux kernel with in-kernel isolation techniques. kSFS repurposes a microkernel-like interface, the FUSE protocol, as a secure interface between the untrusted file system implementations and the remaining kernel components. To realize the approach of sandboxed kernel file systems, several challenges need to be addressed.

Secure In-Kernel Execution. The first challenge is to securely execute untrusted file system code with in-kernel isolation. eBPF, the common framework for sandboxed kernel extensions in Linux, has limited expressiveness, e.g., not allowing loops in general. Thus, applying eBPF as a generic sandboxing mechanism to sandboxing large and complex file systems is still impractical. This is largely caused by its strict guarantees, such as bounded execution time, which is not relevant in file systems. This inspires us to seek an alternative approach that is applicable to generic sandboxing. WebAssembly is a generic sandboxing mechanism, which has been widely deployed in industry, including in security-critical settings [17,53], and generally has good performance. WebAssembly [80] defines a portable code format and the interfaces for executing such code with software-based fault isolation (SFI). Code written in many languages can be compiled to WebAssembly modules with minimal effort. Built WebAssembly modules can then be instantiated as instances running in WebAssembly runtimes, which can either run as standalone programs or be embedded into other programs. Inspired by the success of WebAssembly, we propose using WebAssembly to sandbox kernel file systems in kSFS.

Porting a WebAssembly runtime to kernel space while ensuring security and efficiency is challenging. Existing WebAssembly runtimes, such as WAMR [10] and Wasmtime [11], are usually designed to run in user space. Hence, they depend on user-space APIs, such as signals and pthread, and make important address space assumptions. However, kernel-space programming is significantly different from user-space programming. For example, WebAssembly runtimes usually allocate guard pages and catch out-of-bound memory access with `SIGSEGV` signals, while the Linux kernel does not provide similar interfaces for reserving address space in kernel space or handling kernel page faults. Another difference example is that kernel-space code cannot be "killed", like a user-space program can, so a simple infinite loop can lead to denial-of-service attacks. To generalize a WebAssembly runtime to the kernel, we need to significantly modify its design (§3.3).

Compatibility with Existing Implementations. Another major challenge in implementing sandboxed kernel file systems is to provide a secure and compatible API for sandboxed file system implementations for minimal porting effort. To address this challenge, in addition to the FUSE protocol, we implement a partial WASI [63] interface for file systems to access block devices. Moreover, we provide a modified libfuse library, whose interface is used by most existing FUSE file systems. Using the WASI and libfuse APIs allows kSFS

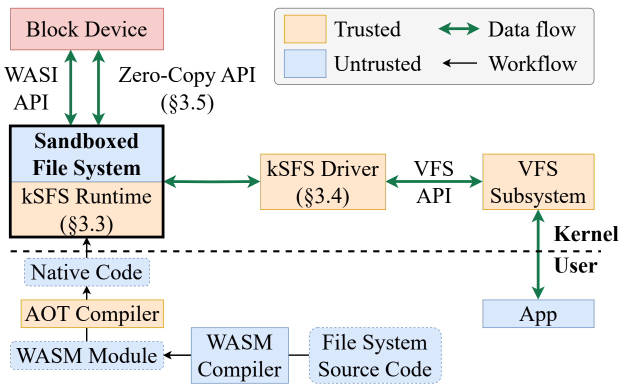

Figure 1: kSFS design overview.

to support existing FUSE file systems with little modification. We discuss how to integrate the compatible interface with WebAssembly file systems in §3.4.

Performance. The last challenge is to ensure that kSFS does not introduce significant performance overhead despite using the FUSE protocol, which is primarily designed for user-space file systems and not for high performance. To address this challenge, we apply several optimizations to kSFS instead of simply adopting the FUSE protocol. For example, we propose an asynchronous zero-copy API, which can avoid buffer copying and allow concurrent file system operations. We discuss such optimizations in §3.5.

3 Design

In this section, we present kSFS's design and explain how it implements secure and efficient in-kernel file systems.

kSFS consists of two major components: an in-kernel WebAssembly runtime that is based on WAMR [10] and the kSFS kernel driver, which is based on the FUSE kernel driver. Fig. 1 shows kSFS's design overview. The kSFS runtime achieves efficient execution of WebAssembly code in the kernel while enforcing security and reliability guarantees. The kSFS kernel driver integrates file systems sandboxed by WebAssembly with the VFS subsystem. In the following sections, we discuss kSFS's system model, threat model, and design in detail.

3.1 System Model

Deploying kSFS requires following an implementation and deployment workflow. To write a sandboxed file system, a developer writes source code using the libfuse API in compatible languages, such as C and C++. Then, the developer compiles the source code with a WebAssembly compiler and links it with kSFS's modified version of libfuse to generate a WebAssembly module. The WebAssembly module is compiled to native machine code with an ahead-of-time (AOT) compiler. The user can load the AOT-compiled module into the kernel and instantiate it as a sandboxed file system instance. From this point on, the sandboxed file system instance interacts with the Linux VFS subsystem via the kSFS kernel driver and, thus, can serve system call requests related to the file system.

3.2 Threat Model

kSFS provides the following security and reliability guarantees: (1) sandboxed file systems can only access the WebAssembly memory and follow strict control flow integrity; (2) sandboxed file systems cannot cause denial-of-service attacks by crashing or stalling the kernel; and (3) sandboxed file systems can only access the files on their own disks. These guarantees are similar to or stronger than prior works on SFI-based kernel isolation [14, 32, 48] or safe file systems [52]. The goal of kSFS is to isolate the kernel from buggy file system implementations, while the correctness of file system implementations, such as data consistency or integrity, is out of the scope of this paper.

kSFS assumes that the WebAssembly runtime and the kSFS driver in the kernel are trusted. It also assumes that the AOT compiler in user space is trusted and that the compiled machine code produced by the AOT compiler is correctly instrumented and not tampered with. This is similar to trusting the kernel compiler. kSFS's design is also compatible with enforcing cryptographic signing of the machine code generated by the AOT compiler if necessary, as the kernel compiler does.

The file system implementation is not trusted and may contain bugs or vulnerabilities, such as arbitrary memory accesses, stack overflows, and control-flow hijacking. For each instance of the file system implementation, kSFS only allows it to access the kernel resources specified by the user, which are usually a single block device, i.e., a partition, for the mount point. kSFS prevents a buggy file system implementation from crashing or stalling the kernel. However, it does not prevent an instance of the file system implementation from causing data loss on its own partition. Users can use separate partitions or disks for isolated failure domains of data loss. Finally, side-channel and hardware attacks, such as Spectre [39] and Rowhammer [38], are out of the scope of this paper, and we expect that existing approaches [9, 54] can help mitigate such attacks.

3.3 WebAssembly Kernel Code Sandboxing

In this section, we explain how kSFS generalizes the WebAssembly runtime to run sandboxed code in kernel space, while satisfying its security and reliability guarantees.

3.3.1 API for WebAssembly Runtime

kSFS adds a new system call to Linux for users to load WebAssembly modules and a kSFS's kernel API to integrate the WebAssembly runtime with the kSFS kernel driver.

System Call API. The user can invoke a new system call introduced by kSFS to execute operations on WebAssembly modules. Through this system call, users can load, unload, and instantiate a WebAssembly module. When loading a WebAssembly module, the user can get a unique module ID (*mid*) that identifies the module. Then, the user can use the system call to instantiate the module to get an *instance file descriptor*, which can be later used for operations on the WebAssembly instance, such as mounting a file system or killing it. The user can close the file descriptor to release a reference to the instance. When instantiating the module, the user can pass command-line arguments, environment variables, and opened file descriptors to the instance. Also, the user can specify the directories where the instance is allowed to open files later. The WebAssembly instance is not allowed to open other files. kSFS currently does not verify WebAssembly code in the kernel and expects that only authorized users can load code that is correctly instrumented by the AOT compiler. Thus, this system call requires root privilege to invoke.

Kernel API. kSFS provides a kernel API for kernel drivers to use the WebAssembly runtime. First, the API allows invoking WebAssembly functions in kernel space, including `malloc` and `free`, which allocate and release memory inside a WebAssembly instance. Second, the API allows validating WebAssembly addresses and converting them to kernel addresses for accessing memory in WebAssembly instances. Third, the API allows raising an exception to a WebAssembly instance to terminate it. Finally, as all the file system implementations used in our evaluation require serialization of file system operations, kSFS currently assumes that only one thread can access a WebAssembly instance simultaneously, and provides a locking API, which internally uses a mutex, to enforce that.

3.3.2 Enforcing Security and Reliability Guarantees

We then discuss how kSFS enforces security and reliability guarantees as shown in §3.2. kSFS implements a WebAssembly runtime based on WAMR. However, WAMR is designed to run in user space and depends on many features only available in user space, such as reserving empty address space and handling signals. This forces us to use different mechanisms in kernel space to achieve the isolation guarantees. We describe how we achieve this in the following paragraphs.

Memory Protection. Similar to user-space WebAssembly runtimes [10,11], kSFS uses binary instrumentation and guard pages for memory protection. kSFS allocates 8 GiB of kernel-space virtual memory for each WebAssembly instance and fills the low virtual memory with physical pages. As a result, any out-of-bound memory access will result in a page fault as the virtual address contains no physical page, which will be caught by kSFS and cause the WebAssembly instance to terminate without crashing the kernel. In §4, we discuss how we achieve this in detail. Other violations, such as insufficient stack space or invalid indirect function calls, will also cause WebAssembly functions to raise exceptions. In this way, kSFS ensures that sandboxed WebAssembly code cannot crash the kernel or access the memory outside its own region.

Stalling Prevention. kSFS prevents WebAssembly code from stalling [71] (i.e., hanging) the kernel by enabling kernel preemption and allowing the termination of an unresponsive WebAssembly instance. In Linux, a thread running in kernel space cannot be forced to terminate by signals, so an infinite loop in kernel code could cause an unkillable process to burn cycles and make other processes fail to be scheduled if the kernel is not preemptible. This would allow a denial-of-service attack by stalling the kernel.To prevent this, we set the kernel preemption model to `PREEMPT`, which allows full preemption in the kernel. This ensures that even if a thread is running WebAssembly code that does not return promptly, other threads can still be scheduled to run. Furthermore, we allow the user to kill a WebAssembly instance via the system call API, which sets a flag in the WebAssembly instance indicating that the instance has been killed. We modified the Linux kernel's interrupt handler so that it checks this flag every time the WebAssembly code is preempted with a timer interrupt, and forces the WebAssembly function to exit when the flag is set by manipulating the registers.

Kernel Resource Access Control. kSFS allows users to apply access control to file systems for kernel resources, such as disks. To achieve this, kSFS implements a partial WASI interface [63] specialized for file system implementations. WASI is a capability-based API for WebAssembly programs to access kernel resources such as files and sockets. The WAMR runtime supports WASI in user space by implementing different backends using OS-dependent interfaces, e.g., POSIX for UNIX-like operating systems. When porting WAMR to the Linux kernel, we added a new backend that directly uses the kernel API to access these kernel resources. Currently, kSFS supports the file system functions of WASI's preview1 version [84]. File systems can then use the POSIX API provided by WASI's compatible layer to access disks. Restricted by WASI's strict capability-based access control, a WebAssembly instance can only access the pre-opened file descriptors or open files from the specified directories, both of which are provided by the user when instantiating the instance. This ensures that each instance of a sandboxed file system can only access the disk specified by the user. kSFS also implements an additional zero-copy API for optimization, which is discussed in §3.5.

3.4 Re-purposing the FUSE Protocol

In this section, we discuss how kSFS re-purposes the FUSE protocol, adopting it as the interface between the kSFS kernel driver and sandboxed file systems. In §6, we discuss in detail how this approach helps deploy file systems to kSFS based on our experience with two real-world file system implementations.

Command Handling. Unlike the FUSE kernel driver, which uses a device file to communicate with the user-space file system implementation for FUSE command execution, the kSFS kernel driver executes FUSE commands with WebAssembly function calls. As all the file system implementations used in our evaluation are ported from single-threaded user-space implementations and require serialization for file system operations, currently, kSFS requires all WebAssembly function calls to be serialized using a per-instance mutex for simplicity. Thus, when multiple threads try to acquire the mutex simultaneously, contention occurs, wasting CPU cycles. Also, invoking WebAssembly functions in different cores increases L1/L2 cache misses. To mitigate the performance impact caused by that, kSFS detects contention before executing each command.

The kSFS kernel driver maintains a request queue for each file system instance, which stores the background and backlogged requests for commands. When contention may happen, i.e., the lock is being held by other threads or the request queue is not empty, the kernel driver will put the request for the command into the request queue instead of directly calling the WebAssembly function. Background commands are always put into the request queue. The kernel driver also maintains a kernel thread for each file system instance that handles the requests in the request queue by calling WebAssembly functions. In this way, kSFS achieves fast synchronous execution of FUSE commands when there is no contention, while avoiding synchronization overhead otherwise.

Data Exchange. kSFS allows file system implementations to exchange data with the kSFS kernel driver to handle FUSE commands. kSFS's WebAssembly runtime protects the kernel memory from WebAssembly instances. Thus, file systems sandboxed by WebAssembly cannot directly read the arguments of FUSE commands or write the results to the kernel memory, making it necessary to copy data, i.e., the arguments or the results serialized according to the FUSE protocol, between the kernel memory and the memory of WebAssembly instances. To achieve data exchange with a WebAssembly instance, the kernel driver allocates a buffer

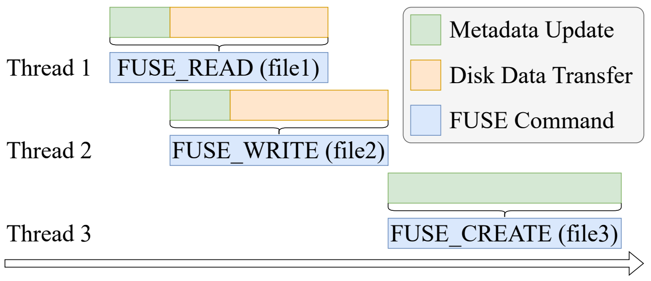

Figure 2: Concurrent operations with the asynchronous zero-copy API.

Table 2: Functions in the synchronous zero-copy API for accessing the zero-copy buffer.

Name	Description
`zc_pread`	transfer data from the disk to the buffer.
`zc_pwrite`	transfer data from the buffer to the disk.
`zc_memcpy`	transfer data between the buffer and WebAssembly memory.
`zc_memset`	fill the buffer with a byte.

inside the memory of the WebAssembly instance. Before executing a command, the kernel driver copies the arguments to the buffer. The WebAssembly function then reads the arguments and writes the results to the buffer before returning to the kernel driver, which copies the results back to the kernel memory after the invocation. To avoid inefficient allocations, the buffer is reused across commands and is only reallocated when the size is smaller than required. kSFS also provides a zero-copy API to avoid costly buffer copies for the `FUSE_READ` and `FUSE_WRITE` commands, which tend to transfer large volumes of data. We discuss the details of this API in §3.5.

3.5 Optimizing File Systems with Zero-Copy API

In this section, we describe how kSFS optimizes kSFS's performance with a zero-copy API. For file read and write operations, the kSFS kernel driver performs bulk data transfer between the kernel memory and the disk, which causes significant overhead if it is implemented with an intermediate buffer in the WebAssembly memory. To avoid this overhead, we carefully optimize kSFS through two mechanisms: a synchronous zero-copy API and an asynchronous zero-copy API.

Synchronous Zero-Copy API. kSFS provides a synchronous zero-copy API for accessing disks by exporting the functions shown in Tab. 2 to WebAssembly instances. For each `FUSE_READ` or `FUSE_WRITE` command, the kSFS kernel driver either gets page cache or user pages that store the data to read or write, or allocates temporary pages for that. We refer to these pages as the zero-copy buffer. When handling a `FUSE_WRITE` command, the WebAssembly function can call `zc_pwrite` to write to the disk with the data from the zero-copy buffer by providing the disk file descriptor, the disk offset, the buffer offset, and the length to write. Then, the kernel function for `zc_pwrite` will write the content of the pages to the disk after validating the arguments. Also, when the WebAssembly function can fallback to call `zc_memcpy` to copy the data from the zero-copy buffer to the WebAssembly memory for further processing when zero-copy disk access is not usable, e.g., due to encryption or compression. When handling a `FUSE_READ` command, the WebAssembly function can similarly call `zc_pread` to transfer the data from the disk to the zero-copy buffer, call `zc_memcpy` to copy the data from the WebAssembly memory to the buffer or call `zc_memset` to fill the zero-copy buffer by providing the offset, byte, and length to fill. All operations to the disk synchronously return to the WebAssembly function after completion.

Asynchronous Zero-Copy API. Many FUSE file systems, including the ones we use in our evaluation in §7, are single-threaded, which significantly limits their performance, especially for workloads heavy on random reads or writes. This drives us to propose an optimized asynchronous zero-copy API. Our optimization is based on the observation that each `FUSE_READ` or `FUSE_WRITE` command can usually be broken down into two steps: the operation on the file system metadata, such as updating file size or allocating clusters, and the actual data transfer between the buffer and the disk. The latter typically takes more time than the operation on the metadata. Thus, by moving it out of the critical section of the WebAssembly instance mutex, more I/O operations can happen concurrently. The asynchronous zero-copy API provides two functions `azc_pread` and `azc_pwrite`, which validate the arguments against the buffer and disk size, append the operation into a queue, and return immediately without actually performing I/O. The kernel driver can then finish the I/O operations after the WebAssembly function returns. Fig. 2 illustrates how the asynchronous zero-copy API achieves concurrent operations by only serializing the update of metadata.

To ensure correctness under concurrency, we carefully design the synchronization protocol. Our approach of applying asynchronous zero-copy API relies on the following assumptions, which hold for many file system implementations, including those used in our evaluation: (1) either only one writing operation or multiple reading operations can happen on one file simultaneously; (2) reading and writing operations to different files are independent, i.e., they can happen simultaneously, as long as the metadata update is serialized; and (3) reading or writing cannot overlap with other operations. These assumptions refer to how file system implementations handle file system operations, but do not impose any semantic constraints on user-space applications. When user-space applications invoke system calls that cause mutually exclusive file system operations, the kSFS kernel driver will synchronize on these operations.

Based on these assumptions, we use a hierarchical locking protocol as follows. The highest-level lock is `inode_lock`, a `rw_semaphore` of the `inode` structure, which is used for enforcing assumption 1. The kSFS kernel driver acquires the lock exclusively for writing and non-exclusively for reading. It is not acquired for other operations. The second-level lock is a `rw_semaphore` for each file system instance, which is used for enforcing assumption 3. The kernel driver acquires the lock non-exclusively for reading or writing and exclusively for other operations. The lowest-level lock is the WebAssembly instance mutex, which is used for enforcing assumption 2. The kernel driver acquires it before each WebAssembly function call and releases it after the call. For operations other than reading or writing, the kernel driver releases the second-level lock immediately after releasing the lowest-level lock. For reading or writing operations, the kernel driver enqueues a work to a workqueue. This work will asynchronously perform the actual I/O operations and release the second-level lock on completion. In this way, kSFS ensures that multiple I/O operations can happen simultaneously and correctly with serialized metadata updates.

4 Implementation

We implemented kSFS by adding two dependencies and the kernel components described in §3, i.e., the kSFS WebAssembly runtime and the kSFS kernel driver, to Linux 6.1.38. The dependencies are the WAMR WebAssembly runtime and the softfp library from glibc, which have 24611 and 8267 LoC, respectively. The kSFS kernel components have 6283 lines of C code. Also, we added 538 lines of C code to libfuse 3.16.1, which can be linked to file system implementations to generate WebAssembly modules. In this section, we discuss the implementation details of kSFS.

AOT Compiler. We modified the AOT compiler in WAMR to support generating code running in kernel space. First, the AOT compiler in WAMR supports generating code that uses addressing related to the `GS` segment to reduce instrumentation overhead [55]. We replaced `GS` with `FS` as the kernel itself uses `GS` in other places. Second, we changed the relocation mode from `LLVMRelocStatic` to `LLVMRelocPIC` to generate position-independent code that can be loaded into the kernel address space without the assumption that the symbols to resolve are in the lower 2 GiB of address space. Generating kernel code also requires some changes to the default command line options. First, as the kernel requires saving and restoring the FPU state for using floating-point instructions, which have high overhead, we disable all these instructions in the compiler and use software floating-point instead. Second, we enable the option for instrumentation of stack space checking in function prologues, which checks if there is enough stack space for the function before executing it and raises a WebAssembly exception if not. In this way, kSFS ensures that WebAssembly code will not cause kernel stack overflows.

Memory Allocation and Protection. kSFS reserves 8 GiB of virtual memory for each WebAssembly instance, following the guard page technique used in user-space runtimes [10, 11]. To achieve this, we added a function, which reserves a virtual memory area in kernel space but only allocates physical pages for part of the area, to the memory management subsystem. kSFS's WebAssembly runtime uses this function to allocate the linear memory of WebAssembly instances. When the WebAssembly instance makes an out-of-bound memory access, a page fault will happen and cause the WebAssembly function to return with an error. Currently, kSFS only considers the x86-64 architecture, which has a large enough kernel address space for allocating 8 GiB of virtual memory. For other architectures that have a smaller kernel address space, such as 32-bit architectures, this approach can be replaced by adding instrumentation of explicit bounds checking, which requires a smaller address space at the cost of higher instrumentation overhead.

WebAssembly Function Invocation. In kSFS, to support isolation and exception handling, calling a WebAssembly function requires additional handling before and after the invocation with a wrapper. Firstly, before calling a WebAssembly function, the wrapper saves the current context to a per-thread buffer, which consists of the instruction pointer, the caller-saved general-purpose registers, and the base address of the `FS` segment. This is similar to `setjmp` in user space. Then, the wrapper sets the base address of the `FS` segment to the base address of the linear memory using the `WRFSBASE` instruction. The native code compiled by the AOT compiler can then use segment-register-based addressing mode to reduce instrumentation overhead. After that, the wrapper calls the actual WebAssembly function, whose prologue first checks the kernel stack to ensure that the remaining stack space is enough. Finally, after calling the WebAssembly function, the wrapper sets the base address of the `FS` segment to the original value before returning.

When a WebAssembly exception happens, the runtime or the compiled native code can then use the saved context to return to the place before calling the WebAssembly function, which is similar to `longjmp`. The wrapper will then return an error. Also, when a WebAssembly function causes a page fault with out-of-bound memory access, the page fault handler will use the saved context to overwrite the registers in the interrupt stack. When the interrupt returns, the execution will then return to the saved place. Moreover, we modified the common exit point of all interrupt handlers (`irqentry_exit`), to add the code for checking if the current thread is invoking a killed WebAssembly instance's function. If so, kSFS will manipulate the registers similarly.

5 Security Analysis

In this section, we analyze how the design of kSFS satisfies the threat model and ensures the security and reliability guarantees as discussed in §3.2.

Memory Isolation and Control Flow Integrity. kSFS enforces strong memory isolation and control flow integrity (CFI) on file system implementations by using WebAssembly instrumentation [81]. Sandboxed file systems can only access their own linear memory allocated at creation. Any out-of-bound access will be caught by the runtime as a page fault and result in the termination of the file system. Moreover, to avoid kernel stack overflow, kSFS checks the stack pointer before invoking any WebAssembly function to ensure enough kernel stack space. With CFI enforced, sandboxed code can only call functions in a pre-defined table with signature verified, thus avoiding hijacking control flow to kernel functions outside the sandbox.

Denial-of-Service Attacks. kSFS prevents denial-of-service attacks such as kernel crashing or stalling. kSFS catches all page faults and WebAssembly exceptions caused by sandboxed file systems. When they happen, kSFS forces the WebAssembly function to return an error and terminates the file system. The kernel will not crash or have corrupted memory. Moreover, kSFS uses kernel preemption to prevent buggy file systems from blocking other threads and allows manually terminating file systems if necessary. This prevents file systems from stalling the kernel.

Device Isolation. kSFS ensures that a file system instance can only access the kernel resources authorized by the user, typically a block device. kSFS achieves this by using the capability-based WASI API, which only allows accessing kernel resources specified at creation. Moreover, kSFS performs the same WASI capability check for the zero-copy APIs. By using kSFS, the user can ensure that the files on different devices are isolated in separate failure domains.

TCB Size. The components kSFS adds to the TCB consist of the WebAssembly runtime and the kSFS driver in the kernel, and the AOT compiler in user space. The size of the TCB kSFS adds to the kernel is 39 LoC, which is smaller than many file systems, thus only slightly increasing the TCB, and can be reused across file systems. For comparison, the eBPF subsystem contains more than 70 kLoC. In addition to the kernel components, the AOT compiler in user space has 87 kLoC and depends on LLVM.

Table 3: Lines of code for NTFS and exFAT's Linux and FUSE implementations and changes made to the FUSE implementations for porting to kSFS in inserted(+)/deleted(-) lines.

File System	Linux	FUSE	kSFS Changes
NTFS	28751	58760	154+/117-
exFAT	7466	5925	104+/ 46-

6 Use Case

In this section, we discuss our experience of building file systems in kSFS by showing two examples of real-world file system implementations that we ported from FUSE to kSFS. Tab. 3 shows the lines of code for the FUSE and Linux implementations, and the changes we made to port them to kSFS. The changes include making them WebAssembly library modules that use kSFS's API to register handlers for FUSE operations, which kSFS invokes, and applying kSFS's zero-copy API for optimization. We ported the FUSE implementations of two real-world file systems NTFS and exFAT into kSFS, both with changes of fewer than 300 LoC, which shows kSFS's deployment ease. We chose these two file systems because their FUSE implementations are well-maintained and widely deployed. For example, currently, the NTFS FUSE implementation is still the default implementation for most Linux distributions.

We ported the FUSE implementations of two file systems to kSFS both using the low-level libfuse API and the asynchronous zero-copy API. For the NTFS file system, we ported a FUSE implementation `lowntfs-3g`, which is part of the NTFS-3G [77] project, to kSFS. We modified the code mainly to apply the asynchronous zero-copy API to the implementation. For the kSFS implementation, we set the WebAssembly linear memory size to 32 MiB. For the exFAT file system, we first wrote a FUSE implementation based on libexfat [60] using the libfuse low-level API. We do not directly use libexfat's default implementation as it uses the high-level API, which causes significant overhead. Then, we ported it to kSFS and applied the asynchronous zero-copy API to it. We set the WebAssembly linear memory size to 256 MiB, which is larger than NTFS. This is because libexfat stores more information in memory than libntfs-3g. Finally, we also wrote a user-space program in 198 LoC, which loads and instantiates the WebAssembly modules of the file systems and mounts them with the modules.

The low effort for porting these two real-world file systems to kSFS shows that kSFS is a general and practical approach to building sandboxed file systems in the Linux kernel. In addition to porting existing file system implementations, developers can also build new ones from scratch taking into account kSFS's optimized API to maximize performance.

7 Evaluation

We evaluate kSFS on two real-world file systems, NTFS and exFAT. We compare their kSFS implementations to other implementations, including highly optimized ones in the Linux kernel and less efficient FUSE and Bento ones, showing that kSFS achieves both security and performance.

7.1 Setup and Methodology

Testbed. We ran all the experiments on an AMD EPYC 7443P server with a 24-core CPU at 2.85 GHz, 128 GiB of RAM, and a Samsung 980 Pro 2TB SSD. The server runs Debian 12 with our patched version of Linux 6.1.38.

File Systems. We use two file systems for evaluation: NTFS and exFAT. Their FUSE implementations were ported to kSFS as mentioned in §6. We compare the kSFS implementations to the Linux and the FUSE ones. To quantify the contributions of each of kSFS's optimizations, we additionally implement an optimized version of the FUSE kernel driver that supports kSFS's asynchronous zero-copy API but does not use WebAssembly in-kernel sandboxing, which we denote as FUSE-zc, and evaluate the performance of these two file system implementations under this optimized version of FUSE. Furthermore, to compare with the approach of using memory-safe languages for file system implementations, we ported Bento [52] to Linux 6.1.38 and made a Bento implementation of exFAT based on a Rust port of libexfat [75]. Also, we run experiments on ext4 as an example of a popular kernel file system for comparison. For performance reasons, we enable write-back cache for all FUSE, FUSE-zc, Bento, and kSFS implementations, as the Linux implementations do. For all experiments, we create a 2 TB partition on the SSD and format it to the specific file system.

Workloads and Measurement Methodology. We use various workloads for evaluation. First, we evaluate kSFS on read/write performance. Second, we evaluate kSFS's performance on the popular file system benchmark tool filebench. Finally, we use real-world applications in different scenarios to evaluate kSFS's performance in such scenarios.

In each experiment, we execute 10 runs for each data point and report the median. We also show the standard deviation for each data point in Tab. 4, and in the figures as error bars.

7.2 Read/Write Performance

We first evaluate kSFS's performance on random read/write operations with the `fio` I/O benchmark tool. We vary the number of threads for `fio` to measure the throughput of the random read/write operations with the block size of 4 KiB under different concurrency. Each experiment lasts 10 seconds on 2 GiB files, with each thread accessing one file. The results

Table 4: Sequential read/write throughput.

		Read (MiB/s)	**Write (MiB/s)**
NTFS	kSFS	2364 ± 58	1220 ± 13
	FUSE	2151 ± 17	1005 ± 18
	FUSE-zc	2385 ± 52	1231 ± 21
	Linux	3325 ± 26	1501 ± 07
exFAT	kSFS	2777 ± 18	1024 ± 139
	FUSE	2414 ± 26	887 ± 86
	FUSE-zc	2769 ± 15	1129 ± 146
	Bento	1028 ± 73	725 ± 03
	Linux	3036 ± 90	878 ± 89
ext4	Linux	3190 ± 73	1533 ± 07

are shown in Fig. 3. The kSFS implementations have significantly better scalability under concurrency than the FUSE and FUSE-zc ones, and have a moderate overhead compared with the Linux ones. For instance, with 64 threads, the kSFS exFAT implementation achieves peak read throughput of 430 kIOPS, which is 31× higher than the FUSE one (13.6 kIOPS) and 2.14× higher than the FUSE-zc one (137 kIOPS), and 23% lower than the Linux one (555 kIOPS) and 31% lower than the Bento one (623 kIOPS) under the same concurrency. This shows that kSFS achieves significant performance improvement over FUSE for concurrent I/O, while still having lower performance than Linux and Bento implementations due to reasons such as instrumentation and protocol overhead. Also, kSFS has significantly better performance than FUSE-zc, which shows that kSFS's fast in-kernel execution is crucial to maximizing the performance of the asynchronous zero-copy I/O.

We then evaluate kSFS's performance on sequential read/write operations with the `fio` I/O benchmark tool. For each implementation, we use `fio` to test the throughput for sequential read and write with a block size of 4 MiB and `fsync` frequency of 64 for write operations, i.e., issuing one `fsync` per 64 write operations. Each experiment lasts 10 seconds on a 128 GiB file with one thread. The results are shown in Tab. 4. The kSFS implementations consistently perform better than the FUSE ones. For instance, the kSFS NTFS implementation achieves 21% higher write throughput than the FUSE one, and is only 19% lower than the Linux NTFS implementation, which has very similar performance to ext4 and offers no isolation. Also, for exFAT, the kSFS implementation has 170% and 41% higher read/write throughput than Bento, respectively, which does not provide isolation except for Rust memory safety checks. The Bento kernel module is forked from the old FUSE kernel module of Linux 4.15, which is less optimized for sequential I/O on large buffers, making the Bento exFAT implementation perform worse than the FUSE one. The kSFS implementations have similar performance to the FUSE-zc implementations. This shows that kSFS's main

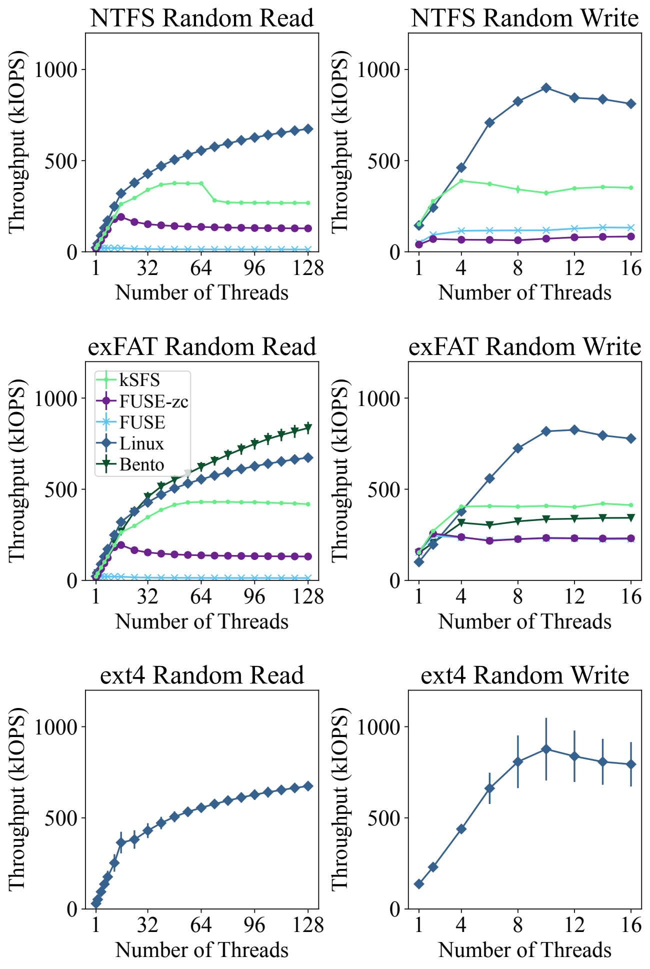

Figure 3: 4 KiB random read/write throughput.

performance improvement for sequential I/O at low IOPS is from asynchronous zero-copy optimization.

7.3 Benchmark Performance

In this section, we evaluate kSFS's performance on the file system benchmark tool filebench [65], which uses synthetic workloads that mimic the real-world settings of server-side applications. We use three example benchmarks: webserver, varmail, and fileserver, all with default parameters. The results are shown in Fig. 4. kSFS achieves significantly better performance than FUSE and FUSE-zc in most cases, except for varmail, where the difference is small considering the higher standard deviation. kSFS has better performance than Bento in two benchmarks, except for the webserver. In the webserver and fileserver benchmarks, FUSE-zc has lower performance than FUSE. This shows that the asynchronous zero-copy API can even reduce the performance of FUSE if not applied together with kSFS's in-kernel execution. In the fileserver benchmark, the tool creates 50 threads, each

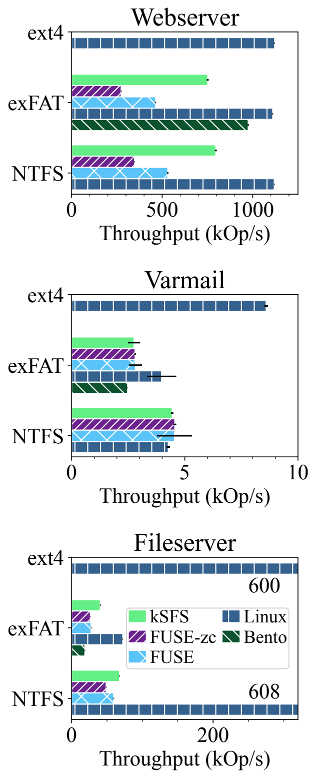

Figure 4: Filebench throughput.

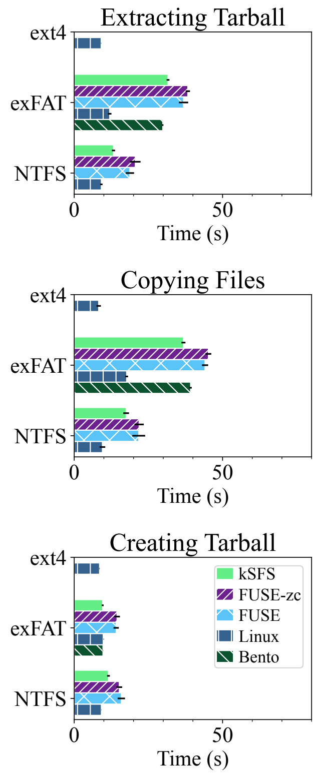

Figure 5: Time for extracting tarball, copying files, and creating tarball.

of which creates a file, writes data to it, reads from it, and finally deletes it. For this benchmark, the kSFS implementations have 14% and 47% higher throughput than the FUSE implementations for NTFS and exFAT, respectively, and 89% and 44% lower throughput than the Linux implementations for NTFS and exFAT, respectively. Notably, the Linux NTFS and ext4 implementations have significantly higher throughput than the kSFS NTFS implementation. This is because the FUSE and kSFS kernel driver writes back the cache of an opened file when closing it, while the Linux NTFS and ext4 implementations do not. As a result, since the file is immediately deleted, the Linux NTFS and ext4 implementations do not write any data to the disk, thus having much higher throughput than other implementations. This optimization for ephemeral files is orthogonal to kSFS's design, and can be ported to kSFS file system implementations if necessary.

7.4 Application Performance

In this section, we evaluate kSFS's performance on real-world applications.

Small Files. We evaluate kSFS's performance on small files, which involves creating, reading, writing, and copying files.

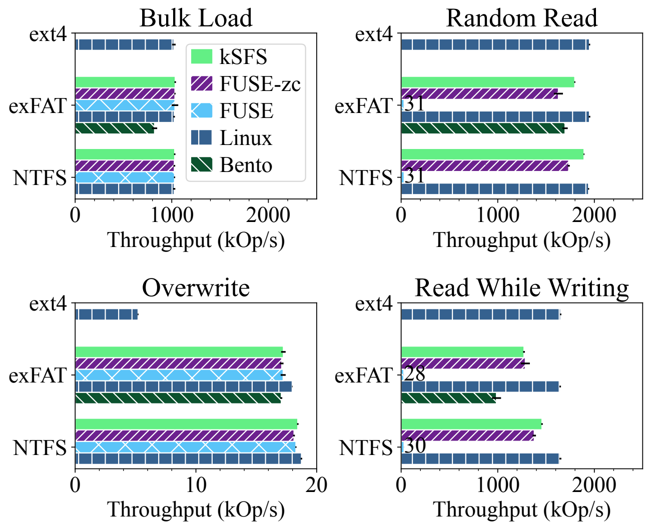

Figure 6: RocksDB throughput. The performance of FUSE file system implementations is particularly low in the benchmarks of random read and read while writing.

We use the following procedure for each experiment and report the time for each step: (1) extracting a tarball, (2) copying the extracted files, and (3) creating a tarball from the new files. We use the Linux kernel source tarball (133 MiB) containing 81,048 files whose total size is 1489 MiB. The results are shown in Fig. 5. The kSFS implementations show significantly better performance than the FUSE ones, by having 25% and 21% lower time on average for NTFS and exFAT, respectively. Also, the kSFS exFAT implementation has similar performance to the Bento one. Compared with the Linux implementations, the kSFS ones have 52% and 90% higher time on average for NTFS and exFAT, respectively. The kSFS and FUSE implementations for exFAT have higher overhead than the ones for NTFS, especially for extracting tarball and copying files. This is mainly due to the inefficiency of libexfat in creating and writing small files. Similar to filebench, FUSE-zc has slightly lower performance than FUSE, which shows the importance of kSFS's in-kernel execution.

RockDB. We evaluate kSFS's performance on RocksDB. We use its official benchmark script with four operations: bulk load, random read, overwrite, and read while writing. We use 100,000,000 keys and 512 MiB of cache in the benchmark. For the benchmark of read while writing, we limit the background write throughput to 100 KiB/s. For other parameters, we use the default values. For example, the number of threads is set to 64. For the last three operations, we run for 60 seconds. The results are shown in Fig. 6. The kSFS implementations show significantly better performance than the FUSE and Bento implementations and moderate overhead compared to the Linux implementations. For example, for random read, the kSFS NTFS implementation achieves

Table 5: Summary of NTFS-3G's CVEs.

Type	CVEs	Example
Buffer Overflow	17	CVE-2022-40284
Out-of-Bound Access	7	CVE-2021-39262
Null Pointer Dereference	1	CVE-2021-39251
Stack Overflow	1	CVE-2021-39257
Memory Exhaustion	1	CVE-2022-30784
Use After Free	1	CVE-2023-52890

a throughput of 1893 kOp/s, which is 60× higher than the FUSE one (31 kOp/s) and 9.1% higher than the FUSE-zc one (1735 kOp/s). This shows that despite still introducing moderate overhead compared with Linux file system implementations, kSFS file system implementations are much more practical for real-world applications than FUSE file systems.

7.5 Security and Completeness

In this section, we evaluate kSFS's security benefits and functionality completeness by analyzing real CVEs and performing compatibility tests.

CVE Analysis. We analyze how kSFS isolates faults and bugs in real-world file systems to prevent kernel crashes and exploits. Tab. 5 summarizes all the 28 CVEs of the NTFS-3g FUSE implementation over the past five years, categorized by type. Of those, 17 CVEs are related to buffer overflows, which can cause crashes or even code execution in the user-space FUSE program. By executing code in WebAssembly sandboxes with kernel resource access control, kSFS isolates such crashes from the rest of the kernel and limits the impact to the mounted disk. Moreover, as WebAssembly protects call stacks and enforces type checks for indirect function invocations, exploiting such stack or heap buffer overflow bugs for code execution is less likely. 9 CVEs are related to out-of-bound access, null pointer dereference, or use after free. kSFS also prevents their propagation to the rest of the kernel. CVE-2021-39257 causes a stack overflow through unbounded recursive function calls. kSFS can prevent such stack overflow by terminating the sandbox, avoiding kernel crashes. CVE-2022-30784 causes memory exhaustion by allocating excessive memory. kSFS ensures that memory exhaustion is limited to the sandbox linear memory and will not impact the kernel allocator or any kernel component beyond the file system. In short, kSFS ensures that when running NTFS-3G within in-kernel WebAssembly sandboxes, none of these CVEs will cause kernel crashes, exploits, or denial of services, nor will they affect any resources other than the mounted disk.

Table 6: Faults injected for testing isolation in kSFS.

Fault	Description
Illegal Access	Access outside linear memory.
Denial-of-Service	Infinite loop.
Stack Overflow	Unbounded recursive function calls.
CPU Exception	Division by zero.

Table 7: POSIX completeness tests in the NTFS implementations.

Implementation	Passed/All Tests
Linux	8777/8789
FUSE	8780/8789
kSFS	8780/8789

Fault Isolation. We then evaluate the effectiveness of kSFS's fault isolation by loading WebAssembly modules with four representative faults, as shown in Tab. 6. The faults include a memory-safety bug, a denial-of-service bug, a stack overflow bug, and a CPU exception bug. We build C programs with these faults into WebAssembly file system modules using the kSFS toolchain and try to mount with these modules. kSFS catches all four faults and recovers from them by terminating the faulty sandboxes promptly, and without kernel crashes, stalls, or any other anomaly. This shows that kSFS effectively prevents the faults described in §3.2 from affecting the kernel.

Completeness. We evaluate kSFS's NTFS functionality completeness using the pjdfstest [20] file system POSIX test suite. We run the test suite on all three NTFS implementations to show that kSFS does not introduce functionality differences when porting file system implementations to the kernel. The results are shown in Tab. 7. The kSFS NTFS implementation passes all the tests that the FUSE implementation passes, and passes three more tests than the Linux implementation. Failed tests are due to subtle semantic differences of hard links between the NTFS implementations and the POSIX standard. This shows that kSFS maintains functionality correctness and completeness of the ported file systems. exFAT is not used in this test, as none of its implementations attempt to comply with POSIX, including the Linux implementation, despite their use in the real-world (e.g., as the default file system required by the SDXC and SDUC standards [61]). Nevertheless, by design, we expect that kSFS exFAT has the same semantics as its FUSE counterpart, as we confirmed with the NTFS pair.

7.6 Evaluation Summary

In summary, we evaluate kSFS's performance on various workloads from microbenchmarks to real-world applications. Also, we evaluate kSFS's security benefits and functionality completeness with real CVEs and tests. kSFS file system implementations consistently show significantly better performance than FUSE implementations. For example, the kSFS NTFS implementation achieves 21% higher sequential write throughput than FUSE, and 19% lower than Linux. For the real-world application RocksDB, the kSFS NTFS implementation achieves 60× higher throughput than FUSE in the random read benchmark, which is only 2.4% lower than Linux. The performance of kSFS file system implementations is moderately lower than Linux implementations, which are highly optimized but lack isolation, while kSFS implementations are ported with minimal effort and provide strong isolation for better security and reliability. Moreover, by adding the asynchronous zero-copy API to the original FUSE and comparing it to kSFS, we show the benefits of kSFS's approach of in-kernel sandboxed execution instead of optimizing FUSE under the user-space architecture.

8 Limitation

Multi-threading Support. kSFS currently only supports single-threaded execution in WebAssembly sandboxes and requires the asynchronous zero-copy API for concurrent operations. Despite the limitation of requiring per-instance mutexes for WebAssembly function calls, our design is compatible with the multi-threading proposal [82] of WebAssembly. We do not anticipate other challenges in supporting multi-threading in kSFS once the proposal is finalized in the WebAssembly specification.

WebAssembly Security. kSFS relies on the correctness of the WebAssembly runtime, which can have bugs and introduces a new attack surface. These bugs can nevertheless be mitigated with testing and verification techniques. For example, WaVe [33] is a verified WebAssembly runtime, which our approach could leverage. In addition to runtime bugs, WebAssembly code does not support many mitigation techniques used by traditional native binaries [41], such as ASLR, page protections, and stack canaries. The lack of mitigation techniques can lead to easier exploits of bugs inside sandboxes, but such exploits will nevertheless be caught and isolated by kSFS without propagating to the rest of the kernel.

9 Discussion

Generalization to Other Kernel Components. We do not make generality claims. However, we note that many operating systems have introduced interfaces for user-space drivers, such as VFIO [72], UMDF [50], and DriverKit [4]. As future work, it could be worth exploring whether our approach of repurposing microkernel-like interfaces is applicable to these interfaces. In addition to secure interfaces, it would require considering other constraints on driver code execution when applying in-kernel sandboxing. For example, Linux requires that interrupt handlers return promptly, while our current approach to stalling prevention, i.e., kernel preemption, does not work in interrupt handlers. A possible approach is to instrument the code to check the execution time. We leave this exploration for future work.

Alternative Isolation Mechanisms. While we currently choose WebAssembly as the isolation mechanism when implementing kSFS, our approach of repurposing microkernel-like interfaces is applicable to other isolation mechanisms such as other SFI implementations or hardware-based isolation. For example, PKS [34] can be used in place of WebAssembly with a similar approach to prior works on user-space in-process isolation using MPK [3, 18, 29, 58, 78]. Moreover, improvements on eBPF [21] that make it more expressive and less constrained might also enable applying eBPF as a generic sandboxing mechanism to our approach.

10 Related Work

User-Space OS Extensions. To achieve better security and reliability and to simplify development, many operating systems have introduced interfaces for extending OS functions in user space [4, 35, 50, 68, 70, 72], which is an approach that draws insights from microkernels [2, 7, 15, 30, 43, 59]. For instance, FUSE [68] is a popular user-space file system API first introduced in Linux. It has been used by many real-world file system implementations and ported to other operating systems [22, 36]. The deployment of user-space OS extensions greatly improves the security and reliability of operating systems. However, they also introduce significant overhead [79]. Some recent works have proposed hiding the latency of mode and context switches by busy waiting [16, 31], which causes a waste of CPU cycles. Compared to user-space OS extensions, kSFS uses in-kernel sandboxed execution to achieve better isolation while minimizing performance overhead.

Kernel Security and Isolation. Many works have been proposed to improve kernel security by testing [23, 25–27, 45, 64, 83, 85], static analysis [5, 6, 24, 47], and isolation [28, 32, 37, 49, 56, 57, 89]. Since its introduction in Linux 3.18, eBPF has been deployed for sandboxed execution across several use cases in the Linux kernel [66, 67, 69]. In addition to these applications of eBPF that have been mainlined into Linux, many research works also use eBPF for various use cases [13, 86, 88, 90], which shows the high demand for sandboxed in-kernel execution for secure customization. However, as the eBPF verifier requires eBPF programs to end in a bounded time, eBPF programs have very limited expressiveness, making it impossible to write drivers totally in eBPF. In addition to the works using eBPF or custom SFI mechanisms, several works [1,40,87] use WebAssembly in kernel space for extending unikernels, optimizing user-space applications, or tracing kernel events. Similar to these works, kSFS introduces WebAssembly into the kernel to support more flexible sandboxed execution for implementing file systems when some guarantees of eBPF are not required.

Memory-Safe Languages in Kernel. Recently, building OS kernels in memory-safe languages has become a hot topic. For instance, Redox [59] and Biscuit [19] are kernels written in Rust and Golang, respectively. Since version 6.1, Linux has supported Rust with the goal of reducing memory bugs. Some works have proposed using Rust to implement file systems [52] and schedulers [51] in Linux. However, as a prior work [42] points out, Rust is not a silver bullet for kernel hardening. Some bugs, such as stack overflow or kernel stalling, cannot be addressed by Rust. Also, writing kernel code in Rust requires high expertise and makes existing code bases not reusable. Compared to the works using Rust, kSFS uses another memory-safe language WebAssembly, which can be the compilation target of many languages. This allows reusing existing code bases written in C. Moreover, kSFS guarantees that sandboxed code cannot crash the kernel or cause denial-of-service attacks, which is not considered in Rust's threat model but is critical to fault isolation for kernel code.

11 Conclusion

We present kSFS, a framework for sandboxing file systems in the Linux kernel that repurposes the FUSE protocol and uses secure in-kernel WebAssembly execution. kSFS achieves practical in-kernel file system sandboxing with strong isolation while introducing moderate performance overhead, which is significantly lower than user-space counterparts but provides similar guarantees. Our evaluation shows that for the real-world application RocksDB, the kSFS implementation of NTFS achieves 60× higher throughput than FUSE in the random read benchmark, which is 2.4% lower than Linux.

Acknowledgments

We would like to express our sincere gratitude to the anonymous reviewers of this and previous submissions and our shepherd for their insightful comments and constructive suggestions, which greatly improved this work. We would also like to thank George Androutsopoulos for his contributions to an earlier prototype. This work was funded in part by gifts from Intel and Google and the NSF awards CNS-2145888 and CNS-2140305.

Ethical Considerations

Potential Stakeholders. We identify the following potential stakeholders: (1) kSFS end users and (2) developers of file systems and the Linux kernel. We analyze the impacts and mitigations to these stakeholders as follows.

kSFS End Users. kSFS provides a new way of adding code to the kernel, which may introduce a new attack interface if users do not understand the correct threat model in which the AoT compiled code is trusted and should be protected from tampering. To mitigate this, we can provide more instructions to users to emphasize the threat model. Also, this can be mitigated by adding support for cryptographic signing verification of loaded code to avoid code tampering.

Kernel and File System Developers. Linux's kernel module mechanism provides different visibility of API to load code of different licenses. However, kSFS provides a way to load code bypassing this mechanism. This can cause legal concerns about the GPL license for kernel and file system developers, including this paper's authors. In our experiments, we use only GPL-licensed code to ensure that we comply with GPL obligations. This concern for developers can be mitigated by enforcing similar license checking to Linux's in kSFS's loading process.

Open Science

Artifact Description

kSFS is bundled with artifacts, including source code and build scripts for all experiments.

How to Access

The artifacts of kSFS can be accessed at `https://github.com/rssys/ksfs-artifacts` or `https://doi.org/10.5281/zenodo.17973523`.

Hardware Dependents

One server with the following hardware is required: at least a 24-core CPU, 128 GiB RAM, and an NVMe SSD.

Setup

Please refer to the `README.md` included in the artifact repository for server setup guidelines.

Experiments

Sequential Read/Write Performance. Run the scripts inside the following directories:

- `experiments/fio-sequential/`
- `experiments/figures/`

`experiments/figures/fio-sequential.py` will print a Latex table as shown in Tab. 4.

Random Read/Write Performance. Run the scripts inside the following directories:

- `experiments/fio-rand/`
- `experiments/figures/`

`experiments/figures/fio-rand.py` will generate `fig-fio-rand.pdf` as shown in Fig. 3.

Filebench Performance. Run the scripts inside the following directories:

- `experiments/filebench/`
- `experiments/figures/`

`experiments/figures/filebench.py` will generate `fig-filebench.pdf` as shown in Fig. 4.

Small File Performance. Run the scripts inside the following directories:

- `experiments/tar/`
- `experiments/figures/`

`experiments/figures/tar.py` will generate `fig-tar.pdf` as shown in Fig. 5.

RocksDB Performance. Run the scripts inside the following directories:

- `experiments/rocksdb/`
- `experiments/figures/`

`experiments/figures/rocksdb.py` will generate `fig-rocksdb.pdf` as shown in Fig. 6.

Alternatively, you can automatically run all the experiments using a single script `experiments/run.sh`. Please refer to `README.md` for more information.

References

[1] Faisal Abdelmonem. Safe kernel extensibility and instrumentation with webassembly. Master's thesis, Carnegie Mellon University, 2025.

[2] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. In *USENIX Summer*, 1986.

[3] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. Kard: lightweight data race detection with per-thread memory protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[4] Apple Inc. . DriverKit | Apple Developer Documentation. https://developer.apple.com/documentation/driverkit, 2024.

[5] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency use-after-free bugs in linux device drivers. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, 2019.

[6] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, page 73–85, New York, NY, USA, 2006. Association for Computing Machinery.

[7] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.

[8] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[9] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[10] Bytecode Alliance. WAMR - WebAssembly Micro Runtime. https://bytecodealliance.github.io/wamr.dev/, 2024.

[11] Bytecode Alliance. Wasmtime. https://wasmtime.dev/, 2024.

[12] Miao Cai, Hao Huang, and Jian Huang. Understanding security vulnerabilities in file systems. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2019.

[13] Xuechun Cao, Shaurya Patel, Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. FetchBPF: Customizable prefetching policies in linux with eBPF. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024.

[14] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.

[15] Haibo Chen, Xie Miao, Ning Jia, Nan Wang, Yu Li, Nian Liu, Yutao Liu, Fei Wang, Qiang Huang, Kun Li, Hongyang Yang, Hui Wang, Jie Yin, Yu Peng, and Fengwei Xu. Microkernel goes general: Performance and compatibility in the HongMeng production microkernel. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

[16] Kyu-Jin Cho, Jaewon Choi, Hyungjoon Kwon, and Jin-Soo Kim. RFUSE: Modernizing userspace filesystem framework through scalable Kernel-Userspace communication. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, 2024.

[17] Cloudflare, Inc. WebAssembly (Wasm) | Cloudflare Workers docs. https://developers.cloudflare.com/workers/runtime-apis/webassembly/, 2024.

[18] Jonathan Corbet. Memory protection keys, 2015. https://lwn.net/Articles/643797/.

[19] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[20] Pawel Jakub Dawidek. pjdfstest – File system test suite. https://github.com/pjd/pjdfstest, 2025.

[21] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. Fast, flexible, and practical kernel extensions. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 2024.

[22] Benjamin Fleischer. macFUSE - File system integration made easy. https://osxfuse.github.io/, 2024.

[23] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. Ski: exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014.

[24] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019.

[25] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.

[26] Sishuai Gong, Dinglan Peng, Deniz Altınbüken, Pedro Fonseca, and Petros Maniatis. Snowcat: Efficient kernel concurrency testing using a learned coverage predictor. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

[27] Sishuai Gong, Wang Rui, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowplow: Effective kernel fuzzing with a learned white-box test mutator. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025.

[28] Google. syzkaller. https://github.com/google/syzkaller/, 2025.

[29] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: intra-process isolation for high-throughput data plane libraries. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, 2019.

[30] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Minix 3: a highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89, July 2006.

[31] Qianbo Huai, Windsor Hsu, Jiwei Lu, Hao Liang, Haobo Xu, and Wei Chen. XFUSE: An infrastructure for running filesystem services in user space. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.

[32] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. KSplit: Automating device driver isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[33] Evan Johnson, Evan Laufer, Zijie Zhao, Dan Gohman, Shravan Narayan, Stefan Savage, Deian Stefan, and Fraser Brown. Wave: a verifiably secure webassembly sandboxing runtime. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[34] Jonathan Corbet. Memory protection keys for the kernel. https://lwn.net/Articles/826554/, 2020.

[35] Antti Kantee. puffs-pass-to-userspace framework file system. In *AsiaBSDCon 2007 Proceedings*, 2007.

[36] Antti Kantee. Refuse : Userspace fuse reimplementation using puffs. In *EuroBSDCon 2007 Proceedings*, 2007.

[37] Arslan Khan, Dongyan Xu, and Dave Jing Tian. Ec: Embedded systems compartmentalization via intra-kernel isolation. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[38] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: an experimental study of dram disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[39] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[40] Martin Kröning, Stefan Lankes, Jonathan Klimt, and Antonello Monti. From browser to kernel: Exploring a lightweight sandboxed approach for unikernel extensions. In *Proceedings of the 13th Workshop on Programming Languages and Operating Systems*, 2025.

[41] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: binary security of webassembly. In *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020.

[42] Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, and Mengwei Xu. An empirical study of Rust-for-Linux: The success, dissatisfaction, and compromise. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024.

[43] J. Liedtke. On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29(5):237–250, December 1995.

[44] Chang Liu, Byungchul Tak, and Long Wang. Understanding performance of ebpf maps. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, 2024.

[45] Congyu Liu, Sishuai Gong, and Pedro Fonseca. Kit: Testing os-level virtualization for functional interference bugs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023.

[46] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013.

[47] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. Dr. checker: a soundy analysis for linux kernel drivers. In *Proceedings of the 26th USENIX Conference on Security Symposium*, 2017.

[48] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[49] Derrick McKee, Yianni Giannaris, Carolina Ortega, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing kernel hacks with hakcs. In *Network and Distributed System Security Symposium 2022 (NDSS 2022)*, 2022.

[50] Microsoft. Overview of UMDF. https://learn.microsoft.com/en-us/windows-hardware/drivers/wdf/overview-of-the-umdf, 2024.

[51] Samantha Miller, Anirudh Kumar, Tanay Vakharia, Ang Chen, Danyang Zhuo, and Thomas Anderson. Enoki: High velocity linux kernel scheduler development. In *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024.

[52] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas Anderson. High velocity kernel file systems with bento. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021.

[53] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[54] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, and Deian Stefan. Swivel: Hardening WebAssembly against spectre. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[55] Shravan Narayan, Tal Garfinkel, Evan Johnson, Zachary Yedidia, Yingchen Wang, Andrew Brown, Anjo Vahldiek-Oberwagner, Michael LeMay, Wenyong Huang, Xin Wang, Mingqiu Sun, Dean Tullsen, and Deian Stefan. Segue & colorguard: Optimizing sfi performance and scalability on modern architectures. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025.

[56] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and vm functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020.

[57] Dinglan Peng, Congyu Liu, Tapti Palit, Pedro Fonseca, Anjo Vahldiek-Oberwagner, and Mona Vij. uswitch: Fast kernel context isolation with implicit context switches. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[58] Dinglan Peng, Congyu Liu, Tapti Palit, Anjo Vahldiek-Oberwagner, Mona Vij, and Pedro Fonseca. Pegasus: Transparent and unified kernel-bypass networking for fast local and remote communication. In *Proceedings of the Twentieth European Conference on Computer Systems*, 2025.

[59] Redox Developers. Redox - Your Next(Gen) OS - Redox - Your Next(Gen) OS. https://www.redox-os.org/, 2024.

[60] relan. Free exFAT file system implementation. https://github.com/relan/exfat, 2023.

[61] SD-3C LLC. Capacity (SD/SDHC/SDXC/SDUC). https://www.sdcard.org/developers/sd-standard-overview/capacity-sd-sdhc-sdxc-sduc/, 2025.

[62] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[63] WASI Subgroup. WebAssembly System Interface. https://github.com/WebAssembly/WASI, 2025.

[64] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. Healer: Relation learning guided kernel fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.

[65] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *login Usenix Mag.*, 41, 2016.

[66] The kernel development community. AF_XDP - The Linux Kernel Documentation. https://docs.kernel.org/networking/af_xdp.html, 2024.

[67] The kernel development community. Extensible Scheduler Class — The Linux Kernel documentation. https://www.kernel.org/doc/html/next/scheduler/sched-ext.html, 2024.

[68] The kernel development community. FUSE - The Linux Kernel Documentation. https://docs.kernel.org/filesystems/fuse.html, 2024.

[69] The kernel development community. LSM BPF Programs — The Linux Kernel documentation. https://docs.kernel.org/bpf/prog_lsm.html, 2024.

[70] The kernel development community. The Userspace I/O HOWTO - The Linux Kernel Documentation. https://docs.kernel.org/driver-api/uio-howto.html, 2024.

[71] The kernel development community. Using RCU's CPU Stall Detector. https://docs.kernel.org/RCU/stallwarn.html, 2024.

[72] The kernel development community. VFIO - "Virtual Function I/O" - The Linux Kernel Documentation. https://docs.kernel.org/driver-api/vfio.html, 2024.

[73] The Mozilla Foundation. WebAssembly - MDN. https://developer.mozilla.org/en-US/docs/WebAssembly, 2024.

[74] The Rust Community. Behavior not considered unsafe - The Rust Reference. https://doc.rust-lang.org/stable/reference/behavior-not-considered-unsafe.html, 2025.

[75] Tomohiro Kusumi. Rust for of libexfat. https://github.com/kusumi/libexfat, 2024.

[76] Linus Torvalds. Re: [PATCH 00/13] [RFC] Rust support. https://lkml.org/lkml/2021/4/14/1099, 2021.

[77] Tuxera. NTFS-3G Safe Read/Write NTFS Driver. https://github.com/tuxera/ntfs-3g, 2023.

[78] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *Proceedings of the USENIX Conference on Security Symposium (USENIX Security)*, 2019.

[79] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of User-Space file systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.

[80] W3C. WebAssembly Core Specification. https://www.w3.org/TR/wasm-core-2/, 2022.

[81] W3C. Security - WebAssembly. https://webassembly.org/docs/security/, 2024.

[82] W3C. Shared-Everything Threads Proposal. https://github.com/WebAssembly/shared-everything-threads, 2025.

[83] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. SyzVegas: Beating kernel fuzzing odds with reinforcement learning. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[84] WASI Subgroup. Legacy WASI docs. https://github.com/WebAssembly/WASI/blob/main/legacy/README.md, 2023.

[85] Tianren Zhang, Sishuai Gong, and Pedro Fonseca. Krr: efficient and scalable kernel record replay. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, 2025.

[86] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[87] Heyang Zhou. kernel-wasm. https://github.com/wasmerio/kernel-wasm, 2020.

[88] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating distributed protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[89] Zhe Zhou, Yanxiang Bi, Junpeng Wan, Yangfan Zhou, and Zhou Li. Userspace bypass: Accelerating syscall-intensive applications. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023.

[90] Tal Zussman, Teng Jiang, and Asaf Cidon. Custom page fault handling with ebpf. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, 2024.