Extension Framework for File Systems in User space

Ashish Bijlani and Umakishore Ramachandran, ***Georgia Institute of Technology***

https://www.usenix.org/conference/atc19/presentation/bijlani

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

Extension Framework for File Systems in User space

Ashish Bijlani
Georgia Institute of Technology

Umakishore Ramachandran
Georgia Institute of Technology

Abstract

User file systems offer numerous advantages over their in-kernel implementations, such as ease of development and better system reliability. However, they incur heavy performance penalty. We observe that existing user file system frameworks are highly general; they consist of a minimal interposition layer in the kernel that simply forwards all low-level requests to user space. While this design offers flexibility, it also severely degrades performance due to frequent kernel-user context switching.

This work introduces EXTFUSE, a framework for developing extensible user file systems that also allows applications to register "thin" specialized request handlers in the kernel to meet their specific operative needs, while retaining the complex functionality in user space. Our evaluation with two FUSE file systems shows that EXTFUSE can improve the performance of user file systems with less than a few hundred lines on average. EXTFUSE is available on GitHub.

1 Introduction

User file systems not only offer better security (i.e., unprivileged execution) and reliability [46] when compared to in-kernel implementations, but also ease the development and maintenance/debugging processes. Therefore, many approaches to develop user space file systems have also been proposed for monolithic Operating Systems (OS), such as Linux and FreeBSD. While some approaches target specific systems [25, 45, 53], a number of general-purpose frameworks for implementing user file systems also exist [3, 14, 29, 33, 47]. FUSE [47], in particular, is the state-of-the-art framework for developing user file systems. Over a hundred FUSE file system have been created in academic/research [11,32,37,41,44,49], as well as in production settings [9, 24, 40, 52].

Being general-purpose, the primary goal of the aforementioned frameworks is to enable easy, yet fully-functional implementation of file systems in user space supporting multiple different functionalities. To do so, they implement a minimal kernel driver that interfaces with the Virtual File System (VFS) operations and simply forwards all low-level requests to user space. For example, when an application (app) makes an `open()` system call, the VFS issues a `lookup` request for each path component. Similarly, `getxattr` requests are issued to read security labels while serving `write()` system calls. Such low-level requests are simply forwarded to user space. This design offers flexibility to developers to easily implement their functionality and apply custom optimizations, but also incurs a high overhead due to frequent user-kernel switching and data copying. For example, despite several recent optimizations, even a simple passthrough FUSE file system can introduce up to 83% overhead on an SSD [50]. As a result, some FUSE file systems have been replaced by alternative implementations in production [8, 22, 24].

There have been attempts to address performance issues of user file system frameworks, for example, by eliminating user-kernel switching in FUSE under certain scenarios [28, 34]. Nevertheless, the optimizations proposed pertain to their specific use cases and do not address the inherent design limitations of existing frameworks.

We observe that the interfaces exported by existing user file system frameworks are too low-level and general-purpose. As such, they fail to match the specific operative needs of file systems. For example, `getxattr` requests during `write()` can be completely eliminated for files containing no security labels. `lookup` replies from the daemon could be cached and validated in the kernel to reduce context switches to user space. Similarly, when stacking sandboxing functionality for enforcing custom permissions checks in `open()` system call, I/O requests (e.g., `read/write`) could be passed directly through the host file system. Nevertheless, modifying existing frameworks to efficiently address specific functional and performance requirements of each use case is impractical.

We borrow the idea of safely extending system services at runtime from past works [6, 13, 43, 56] and propose to address the performance issues in existing user file systems frameworks by allowing developers to safely extend the functionality of the kernel driver at runtime for specialized handling of their use case. This work introduces EXTFUSE, an extension framework for file systems in user space that allows developers to define specialized "thin" *extensions* along with auxiliary data structures for handling low-level requests in the kernel. The extensions are safely executed under a sandboxed runtime environment in the kernel immediately as the requests are issued from the upper file system layer (e.g., VFS), thereby offering a fine-grained ability to either serve each request entirely in the kernel (*fast path*) or fall back to the existing complex logic in user space (*slow path*) to achieve the desired operative goals of functionality and performance. The fast and slow paths can access (and modify) the auxiliary data structures to define custom logic for handling requests.

EXTFUSE consists of three components. First, a helper user library that provides a familiar set of file system APIs to register extensions and implement custom fast-path functionality in a subset of the C language. Second, a wrapper (no-ops) interposition driver that bridges with the low-level VFS interfaces and provides the necessary support to forward requests to the registered kernel extensions as well as to the lower file system, as needed. Third, an in-kernel Virtual Machine (VM) runtime that safely executes the extensions.

We have built EXTFUSE to work in concert with existing user file system frameworks to allow both the fast and the existing slow path to coexist with no overhauling changes to the design of the target user file system. Although there are several user file system frameworks, this work focuses only on FUSE because of its wide-spread use. Nonetheless, we have implemented EXTFUSE as in a modular fashion so it can be easily adopted for others.

We added support for EXTFUSE in four popular FUSE file systems, namely LoggedFS [16], Android sdcard daemon, MergerFS, and BindFS [35]. Our evaluation of the first two shows that EXTFUSE can offer substantial performance improvements of user file systems by adding less than a few hundred lines, on average.

This paper makes the following contributions:

- We identify optimization opportunities in user file system frameworks for monolithic OSes (§2) and propose extensible user file systems.
- We present the design (§3) and architecture (§3.4) of EXTFUSE, an extension framework for user file systems that offers the performance of kernel file systems, while retaining the safety properties of user file systems.
- We demonstrate the applicability of EXTFUSE by adopting it for FUSE, the state-of-the-art user file system framework, and evaluating it on Linux (§6).
- We show the practical benefits and limitations of the EXTFUSE with two popular FUSE file systems, one deployed in production (§6.2).

2 Background and Extended Motivation

This section provides a brief technical background on FUSE and its limitations that motivate our work.

2.1 FUSE

FUSE is the state-of-the-art framework for developing user file systems. It consists of a loadable kernel driver and a helper user-space library that provides a set of portable APIs to allow users to implement their own fully-functional file system as an unprivileged daemon process on Unix-based systems with no additional kernel support. The driver is a simple interposition layer that only serves as a communication channel between the user-space daemon and the VFS. It registers a new file system to interface with the VFS operations and directly forwards all low-level requests to the daemon, as received.

The library provides two different sets of APIs. First, a `fuse_lowlevel_ops` interface that exports all VFS operations such as `lookup` for path to inode mapping. It is used by file systems that need to access low-level abstractions (e.g., inodes) for custom optimizations. Second, a high-level `fuse_operations` interface that builds on the low-level APIs. It hides complex abstractions and offers a simple (e.g., path-based) API for the ease of development. Depending on their particular use case, developers can adopt either of the two APIs. Furthermore, many operations in both APIs are optional. For example, developers can choose to not handle extended attributes (xattrs) operations (e.g., `getxattr`).

As apps make system calls, the VFS layer forwards all low-level requests to the kernel driver. For example, to serve the `open()` system call on Linux, the VFS issues multiple `lookup` requests to the driver, one for each input path component. Similarly, every `write()` request is preceded by a `getxattr` request from the VFS to fetch the security labels. The driver queues up all requests, along with the relevant parameters, for the daemon through `/dev/fuse` device file and blocks the calling app thread until the requests are served. The daemon, through the `libfuse` interface, retrieves the requests from the queue, processes them as needed, and enqueues the results for the driver to read. The driver copies the results, populating VFS caches, as appropriate (e.g., page cache for `read`/`write`, dcache for dir entries from `lookup`), and wakes the app thread and returns the results to it. Once cached, the subsequent accesses to the same data are served with no user-kernel round-trip communication.

In addition to the common benefits of user file systems such as ease of development and maintenance, FUSE also provides app-transparency and fine-grained control to developers over the low-level API to easily support their custom functionality. Furthermore on Linux, the FUSE driver is GPL-licensed, which detaches the implementation in user space from legal obligations [57]. It also supports multiple language bindings (e.g., Python, Go, etc.), thereby enabling access to ecosystem of third-party libraries for reuse.

Given its general-purpose design and numerous advantages, over a hundred FUSE file systems with different functionalities have been created. A large majority of them are stackable; that is, they introduce incremental functionality on the host file system [38]. FUSE has long served as a popular tool for quick experimentation and prototyping new file systems in academic and research settings [11, 32, 37, 41, 44, 49]. However, more recently a host of FUSE file systems have also been deployed in production. Both Gluster [40] and Ceph [52] cluster file systems use a FUSE network client implementation. Android v4.4 introduced FUSE sdcard daemon (stackable) to add multi-user support and emulate FAT functionality on the EXT4 file system [24].

While exporting low-level abstractions and VFS interfaces offers more control and flexibility to developers, this design comes with a cost. It induces frequent user-kernel round-

Media	W/ xattr (%Diff)		W/O xattr (%Diff)	
	SW	RW	SW	RW
HDD	-0.29	-3.80	-0.17	-2.83
SSD	-11.5	-23.38	+0.31	-12.24

Table 1: Percentage difference between I/O throughput (ops/sec) for EXT4 vs FUSE StackfsOpt (see §6) under single thread 4K Seq Write (SW), and Rand Write (RW) settings on a 60GB file across different storage media as reported by Filebench [48]. Received 15 million `getxattr` requests.

trip communication and data copying, and thus inevitably yields poor runtime performance. Nonetheless, FUSE has evolved significantly over the years; several optimizations have been added to minimize the user-kernel communication: zero-copy data transfer (splicing), and utilizing system-wide shared VFS caches (page cache for data I/O and dentry cache for metadata). However, despite these optimizations, FUSE severely degrades runtime performance in a host of scenarios. For instance, data caching improves I/O throughput by batching requests (e.g., read-aheads, small writes), but it does not help apps that perform random reads or demand low write latency (e.g., databases). Splicing is only used for over 4K requests. Therefore, apps with small writes/reads (e.g., Android apps, etc.) will incur data copying overheads. Worse yet, the overhead is higher with faster storage media. Even a simple passthrough (no-ops) FUSE file system can introduce up to 83% overhead for metadata-heavy workloads on SSD [50].

The performance penalty incurred by user file systems overshadows their benefits. Consequently, some user file systems have been replaced with alternative implementations in production. For instance, Android v7.0 replaced the sdcard daemon with its in-kernel implementation after several years [24]. Ceph [52] adopted an in-kernel client for Linux kernel [8].

2.2 Generality vs Specialization

We observe that being highly general-purpose, the FUSE framework induces unnecessary user-kernel communication in many cases, yielding low throughput and high latency. For instance, `getxattr` requests generated by the VFS during `write()` system calls (one per `write`) can double the number of user-kernel transitions, which decreases the I/O throughput of sequential writes by over 27% and random writes by over 44% compared to native (EXT4) performance (Table 1). Moreover, the penalty incurred is higher with the faster media.

Nevertheless, simple filtering or caching metadata replies in the kernel can substantially reduce user-kernel communication. For instance, by caching the last `getxattr` reply in the FUSE driver and simply validating the cached state for every subsequent `getxattr` request from the VFS on the same file, unnecessary user-kernel transitions can be eliminated to achieve significant improvement in `write` performance. Similarly, replies from other metadata operations, such as `lookup` and `getattr` can be cached, validated, and served entirely within the kernel. Note that custom validation of cached metadata is imperative, and lack of support to do so may result in incorrect behavior as happens in the case of optimized FUSE that leverages VFS caches to serve requests (§5.1).

Many stackable user file systems add a thin layer of functionality; they perform simple checks in a few operations and pass remaining requests directly through the host (lower) file system. LoggedFS [16] filters requests that must be logged and do so by accessing host file system services. Union file systems such as MergerFS [20] determine the backend host file in `open()` and redirects I/O requests to it. Android sdcard daemon performs access permission checks only in metadata operations (e.g., `open`, `lookup`), but data I/O requests (e.g., `read`, `write`, etc.) are simply forwarded to the lower file system. Thin functionality that realizes such use cases does not need any complex processing in user space, and therefore can easily be stacked in the kernel, thereby avoiding expensive user-kernel switching to yield lower latency and higher throughput for the same functionality. Furthermore, data I/O requests could be directly forwarded to the host file system.

The FUSE framework offers a few configuration (config) options to the developers to tune the behavior of its kernel driver for their use case. However, those options are coarse-grained and implemented as fixed checks embedded in the driver code; thus, it offers limited static control. For example, for file systems that do not support certain operations (e.g., `getxattr`), the FUSE driver caches this knowledge upon first `ENOSUPPORT` reply and does not issue such requests subsequently. While this benefits the file systems that completely omit certain functionality (e.g., security xattr labels), it does not allow custom and fine-grained filtering of requests that may be desired by some file systems supporting only partial functionality. For example, file systems providing encryption (or compression) functionality may desire a fine-grained custom control over the kernel driver to skip the decryption (or decompression) operation in user space during `read()` requests on non-sensitive (or unzipped) files.

As such, the FUSE framework proves to be too low-level and general-purpose in many cases. While it enables a number of different use cases, modifying the framework to efficiently handle special needs of each use case is impractical. This is a typical unbalanced generality vs. specialization problem, which can be addressed by extending the functionality of the FUSE driver in the kernel [6, 13, 43].

3 Design

In this section, we 1) present an overview of ExtFUSE, 2) discuss the design goals and challenges we faced, and 3) mechanisms we adopted to address those challenges.

3.1 Overview

ExtFUSE is a framework for developing extensible FUSE file systems for UNIX-like monolithic OSes. It allows the

unprivileged FUSE daemon processes to register "thin" extensions in the kernel for specialized handling of low-level file system requests, while retaining their existing complex logic in user space to achieve the desired level of performance.

The registered extensions are safely executed under a sandboxed eBPF runtime environment in the kernel (§3.3), immediately as requests are issued from the upper file system (e.g., VFS). Sandboxing enables the FUSE daemon to safely extend the functionality of the driver at runtime and offers a fine-grained ability to either serve each request entirely in the kernel or fall back to user space, thereby offering safety of user space and performance of kernel file systems.

ExtFUSE also provides a set of APIs to create shared key-value data structures (called maps) that can host abstract data blobs. The user-space daemon and its kernel extensions can leverage maps to store/manipulate their custom data types as needed to work in concert and serve file system requests in the kernel without incurring expensive user-kernel round-trip communication if deemed unnecessary.

3.2 Goals and Challenges

The over-arching goal of ExtFUSE is to carefully balance the safety and runtime extensibility of user file systems to achieve the desired level of performance and specialized functionality. Nevertheless, in order for developers to use ExtFUSE, it must also be easy to adopt. We identify the following concrete design goals.

Design Compatibility. The abstractions and interfaces offered by ExtFUSE framework must be compatible with FUSE without hindering existing functionality or properties. It must be general-purpose so as to support multiple different use cases. Developers must further be able to adopt ExtFUSE for their use case without overhauling changes to their existing design. It must be easy for them to implement specialized extensions with little to no knowledge of the underlying implementation details.

Modular Extensibility. ExtFUSE must be highly modular and limit any unnecessary new changes to FUSE. Particularly, developers must be able to retain their existing user-space logic and introduce specialized extensions only if needed.

Balancing Safety and Performance. Finally with ExtFUSE, even unprivileged (and untrusted) FUSE daemon processes must be able to safely extend the functionality of the driver as needed to offer performance that is as close as possible to the in-kernel implementation. However, unlike Microkernels [2, 23, 30] that host system services in separate protection domains as user processes or the OSes that have been developed with safe runtime extensibility as a design goal [6, 13], extending system services of general-purpose UNIX-like monolithic OSes poses a design trade-off question between the safety and performance requirements.

Untrusted extensions must be as lightweight as possible, with their access restricted to only a few well-defined APIs to guarantee safety. For example, kernel file systems offer near-native performance, but executing complex logic in the kernel results in questionable reliability. Additionally, most OS kernels employ Address Space Randomization [36], Data Execution Prevention [5], etc. for code protection and hiding memory pointers. Providing unrestricted kernel access to extensions can render such protections useless. Therefore, extensions must not be able to access arbitrary memory addresses or leak pointer values to user space.

However, severely restricting extensibility can prevent user file systems from fully meeting their operative performance and functionality goals, thus defeating the purpose of extensions in the first place. Therefore, ExtFUSE must carefully balance safety and performance goals.

Correctness. Furthermore, specialized extensions can alter the existing design of user file systems, which can lead to correctness issues. For example, there will be two separate paths (fast and slow) both operating on the requests and data structs at the same time. The framework must provide a way for them to synchronize and offer safe concurrent accesses.

3.3 eBPF

ExtFUSE leverages extended BPF (eBPF) [26], an in-kernel Virtual Machine (VM) runtime framework to load and safely execute user file system extensions.

Richer functionality. eBPF is an extension of classic Berkeley Packet Filters (BPF), an in-kernel interpreter for a pseudo machine architecture designed to only accept simple network filtering rules from user space. It enhances BPF to include more versatility, such as 64-bit support, a richer instruction set (e.g., call, cond jump), more registers, and native performance through JIT compilation.

High-level language support. The eBPF bytecode backend is also supported by Clang/LLVM compiler toolchain, which allows functionality logic to be written in a familiar high-level language, such as C and Go.

Safety. The eBPF framework provides a safe execution environment in the kernel. It prohibits execution of arbitrary code and access to arbitrary kernel memory regions; instead, the framework restricts access to a set of kernel helper APIs depending on the target kernel subsystem (e.g., network) and required functionality (e.g., packet handling). The framework includes a static analyzer (called verifier) that checks the correctness of the bytecode by performing an exhaustive depth-first search through its control flow graph to detect problems, such as infinite loops, out-of-bound, and illegal memory errors. The framework can also be configured to allow or deny eBPF bytecode execution request from unprivileged processes.

Key-Value Maps. eBPF allows user space to create *map* data structures to store arbitrary key-value blobs using system calls and access them using file descriptors. Maps are also accessible to eBPF bytecode in the kernel, thus providing a communication channel between user space and the bytecode

to define custom key-value types and share execution state or data. Concurrent accesses to maps are protected under read-copy update (RCU) synchronization mechanism. However, maps consume unswappable kernel memory. Furthermore, they are either accessible to everyone (e.g., by passing file descriptors) or only to `CAP_SYS_ADMIN` processes.

eBPF is a part of the Linux kernel and is already used heavily by networking, tracing, and profiling subsystems. Given its rich functionality and safety properties, we adopt eBPF for providing support for extensible user file systems. Specifically, we define a white-list of kernel APIs (including their parameters and return types), and abstractions that user file system extensions can safely use to realize their specialized functionality. The eBPF verifier utilizes the whitelist to validate the correctness of the extensions. We also build on eBPF abstractions (e.g., maps) and apply further access restrictions to enable safe in-kernel execution, as needed.

3.4 Architecture

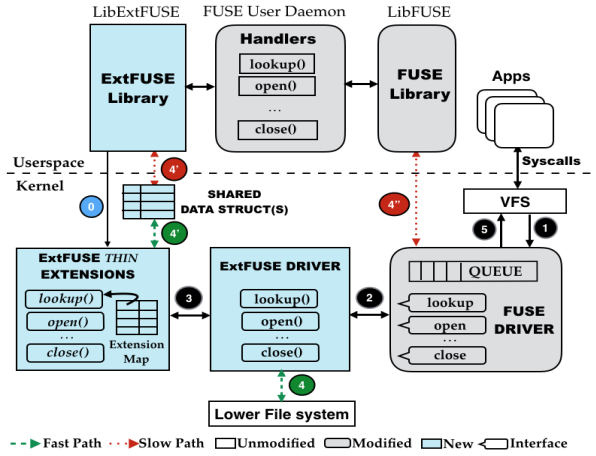

Figure 1: Architectural view of the ExtFUSE framework. The components modified or introduced have been highlighted.

Figure 1 shows the architecture of the ExtFUSE framework. It is enabled by three core components, namely a kernel file system (`driver`), a user library (`libExtFUSE`), and an in-kernel eBPF virtual machine runtime (`VM`).

The ExtFUSE driver uses interposition technique to interface with FUSE at low-level file system operations. However, unlike the FUSE driver that simply forwards file system requests to user space, the ExtFUSE driver is capable of directly delivering requests to in-kernel handlers (extensions). It can also forward a few restricted set of requests (e.g., `read`, `write`) to the host (lower) file system, if present. The latter is needed for stackable user file systems that add thin functionality on top of the host file system. `libExtFUSE` exports a set of APIs and abstractions for serving requests in the kernel, hiding the underlying implementation details.

Use of `libExtFUSE` is optional and independent of `libfuse`. The existing file system handlers registered with `libfuse` continue to reside in user space. Therefore, their invocation incurs context switches, and thus, we refer to their execution as the *slow* path. With ExtFUSE, user space can also register kernel extensions that are invoked immediately as file system requests are received from the VFS in order to allow serving them in the kernel. We refer to the in-kernel execution as the *fast* path. Depending upon the return values from the fast path, the requests can be marked as served or be sent to the user-space daemon via the slow path to avail any complex processing as needed. Fast path can also return a special value that instructs the ExtFUSE driver to interpose and forward the request to the lower file system. However, this feature is only available to stackable user file systems and is verified when the extensions are loaded in the kernel.

FS Interface	API(s)	Abstractions	Description
Low-level	`fuse_lowlevel_ops`	`Inode`	FS Ops
Kernel Access	**API(s)**	**Abstractions**	**Description**
eBPF Funcs	`bpf_*`	`UID, PID, etc.`	Helper Funcs
FUSE	`extfuse_reply_*`	`fuse_reply_*`	Req Output
Kernel	`bpf_set_pasthru`	`FileDesc`	Enable Pthru
Kernel	`bpf_clear_pasthru`	`FileDesc`	Disable Pthru
DataStructs	**API(s)**	**Abstractions**	**Description**
SHashMap	`CRUD`	`Key/Val`	Hosts arbitrary data blobs
InodeMap	`CRUD`	`FileDesc`	Hosts upper-lower inode pairs

Table 2: APIs and abstractions provided by ExtFUSE. It provides FUSE-like file system interface for easy portability. CRUD (create, read, update, and delete) APIs are offered for map data structures to operate on Key/Value pairs. Kernel accesses are restricted to standard eBPF kernel helper functions. We introduced APIs to access the same FUSE request parameters as available to user space.

The fast path interfaces exported by `libExtFUSE` are the same as those exported by `libfuse` to the slow path. This is important for easy transfer of design and portability. We leverage eBPF support in the LLVM/Clang compiler toolchain to provide developers with a familiar set of APIs and allow them to implement their custom functionality logic in a subset of the C language.

The extensions are loaded and executed inside the kernel under the eBPF VM sandbox, thereby providing user space a fine-grained ability to safely extend the functionality of FUSE kernel driver at runtime for specialized handling of each file system request.

3.5 ExtFUSE APIs and Abstractions

`libExtFUSE` provides a set of high-level APIs and abstractions to the developers for easy implementation of their specialized extensions, hiding the complex implementation details. Table 2 summarizes the APIs. For handling file system operations, `libExtFUSE` exports the familiar set of FUSE interfaces and corresponding abstractions (e.g., inode) for design compatibility. Both low-level as well as high-level file system interfaces are available, offering flexibility and development ease. Furthermore, as with `libfuse`, the daemon can reg-

ister extensions for a few or all of the file system APIs, offering them flexibility to implement their functionality with no additional development burden. The extensions receive the same request parameters (`struct fuse_[in,out]`) as the user-space daemon. This design choice not only conforms to the principle of least privilege, but also offers the user-space daemon and the extensions the same interface for easy portability.

For hosting/sharing data between the user daemon and kernel extensions, `libExtFUSE` provides a secure variant of eBPF `HashMap` key/value data structure called `SHashMap` that stores arbitrary key/value blobs. Unlike regular eBPF maps that are either accessible to all user processes or only to `CAP_SYS_ADMIN` processes, `SHashMap` is only accessible by the unprivileged daemon that creates it. `libExtFUSE` further abstracts low-level details of `SHashMap` and provides high-level `CRUD` APIs to create, read, update, and delete entries (key/value pairs).

ExtFUSE also provides a special `InodeMap` to enable passthrough I/O feature for stackable ExtFUSE file systems (§5.2). Unlike `SHashMap` that stores arbitrary entries, `InodeMap` takes open file handle as key and stores a pointer to the corresponding lower (host) inode as value. Furthermore, to prevent leakage of inode object to user space, the `InodeMap` values can only be read by the ExtFUSE driver.

3.6 Workflow

To understand how ExtFUSE facilitates implementation of extensible user file systems, we describe its workflow in detail. Upon mounting the user file system, FUSE driver sends `FUSE_INIT` request to the user-space daemon. At this point, the user daemon checks if the OS kernel supports ExtFUSE framework by looking for `FUSE_CAP_ExtFUSE` flag in the request parameters. If supported, the daemon must invoke `libExtFUSE init` API to load the eBPF program that contains specialized handlers (extensions) into the kernel and register them with the ExtFUSE driver. This is achieved using `bpf_load_prog` system call, which invokes eBPF verifier to check the integrity of the extensions. If failed, the program is discarded and the user-space daemon is notified of the errors. The daemon can then either exit or continue with default FUSE functionality. If the verification step succeeds and the JIT engine is enabled, the extensions are processed by the JIT compiler to generate machine assembly code ready for execution, as needed.

Extensions are installed in a `bpf_prog_type` map (called *extension map)*, which serves effectively as a jump table. To invoke an extension, the FUSE driver simply executes a `bpf_tail_call` (far jump) with the FUSE operation code (e.g., `FUSE_OPEN`) as an index into the extension map. Once the eBPF program is loaded, the daemon must inform ExtFUSE driver about the kernel extensions by replying to `FUSE_INIT` containing identifiers to the extension map.

Once notified, ExtFUSE can safely load and execute the extensions at runtime under the eBPF VM environment. Every request is first delivered to the fast path, which may decide to 1) serve it (e.g., using data shared between the fast and slow paths), 2) pass the request through to the lower file system (e.g., after modifying parameters or performing access checks), or 3) take the slow path and deliver the request to user space for complex processing logic (e.g., data encryption), as needed. Since the execution path is chosen per-request independently and the fast path is always invoked first, the kernel extensions and user daemon can work in concert and synchronize access to requests and shared data structures. It is important to note that the ExtFUSE driver only acts as a thin interposition layer between the FUSE driver and kernel extensions, and in some cases, between the FUSE driver and the lower file system. As such, it does not perform any I/O operation or attempts to serve requests on its own.

Component	Version	Loc Modified	Loc New
FUSE kernel driver	4.11.0	312	874
FUSE user-space library	3.2.0	23	84
ExtFUSE user-space library	-	-	581

Table 3: Changes made to the existing Linux FUSE framework to support ExtFUSE functionality.

4 Implementation

To implement ExtFUSE, we provided eBPF support for FUSE. Specifically, we added additional kernel helper functions and designed two new map types to support secure communication between the user-space daemon and kernel extensions, as well as support for passthrough access in read/write. We modified FUSE driver to first invoke registered eBPF handlers (extensions). Passthrough implementation is adopted from WrapFS [54], a wrapper stackable in-kernel file system. Specifically, we modified FUSE driver to pass I/O requests directly to the lower file system.

Since with ExtFUSE developers can install extensions to bypass the user-space daemon and pass I/O requests directly to the lower file system, a malicious process could stack a number of ExtFUSE file systems on top of each other and cause the kernel stack to overflow. To guard against such attacks, we limit the number of ExtFUSE layers that could be stacked on a mount point. We rely on `s_stack_depth` field in the super-block to track the number of stacked layers and check it against `FILESYSTEM_MAX_STACK_DEPTH`, which we limit to two. Table 3 reports the number of lines of code for ExtFUSE. We also modified `libfuse` to allow apps to register kernel extensions.

5 Optimizations

Here, we describe a set of optimizations that can be enabled by leveraging custom kernel extensions in ExtFUSE to implement in-kernel handling of file system requests.

```
1  void handle_lookup(fuse_req_t req, fuse_ino_t pino,
2              const char *name) {
3      /* lookup or create node @cname parent @pino */
4      struct fuse_entry_param e;
5      if (find_or_create_node(req, pino, name, &e)) return;
6  +   lookup_key_t key = {pino, name};
7  +   lookup_val_t val = {0/*not stale*/, &e};
8  +   extfuse_insert_shmap(&key, &val); /* cache this entry */
9      fuse_reply_entry(req, &e);
10 }
```

Figure 2: FUSE daemon `lookup` handler in user space. With EXTFUSE, lines 6-8 (+) enable caching replies in the kernel.

```
1  int lookup_extension(extfuse_req_t req, fuse_ino_t pino,
2              const char *name) {
3      /* lookup in map, bail out if not cached or stale */
4      lookup_key_t key = {pino, name};
5      lookup_val_t *val = extfuse_lookup_shmap(&key);
6      if (!val || atomic_read(&val->stale)) return UPCALL;
7      /* EXAMPLE: Android sdcard daemon perm check */
8      if (!check_caller_access(pino, name)) return -EACCES;
9      /* populate output, incr count (used in FUSE_FORGET) */
10     extfuse_reply_entry(req, &val->e);
11     atomic_incr(&val->nlookup, 1);
12     return SUCCESS;
13 }
```

Figure 3: EXTFUSE `lookup` kernel extension that serves valid cached replies, without incurring any context switches. Customized checks could further be included; Android sdcard daemon permission check is shown as an example (see Figure 10).

5.1 Customized in-kernel metadata caching

Metadata operations such as `lookup` and `getattr` are frequently issued, and thus form high sources of latency in FUSE file systems [50]. Unlike VFS caches that are only reactive and fixed in functionality, EXTFUSE can be leveraged to proactively cache metadata replies in the kernel. Kernel extensions can be installed to manage and serve subsequent operations from caches without switching to user space.

Example. To illustrate, let us consider the `lookup` operation. It is the most common operation issued internally by the VFS for serving `open()`, `stat()`, and `unlink()` system calls. Each component of the input path string is searched using `lookup` to fetch the corresponding inode data structure. Figure 2 lists code fragment for FUSE daemon handler that serves `lookup` requests in user space (slow path). The FUSE `lookup` API takes two input parameters: the parent node ID and the next path component name. The node ID is a 64-bit integer that uniquely identifies the parent inode. The daemon handler function traverses the parent directory, searching for the child entry corresponding to the next path component. Upon successful search, it populates the `fuse_entry_param` data structure with the node ID and attributes (e.g., size) of the child, and sends it to the FUSE driver, which creates a new inode for the dentry object representing the child entry.

With EXTFUSE, developers could define a `SHashMap` that hosts `fuse_entry_param` replies in the kernel (lines 7-10). A composite key generated from the parent node identifier and the next path component string arguments is used as an index into the map for inserting corresponding replies. Since the map is also accessible to the extensions in the kernel, subsequent requests could be served from the map by installing the EXTFUSE `lookup` extension (fast path). Figure 3 lists its code fragment. The extension uses the same composite key as an index into the hash map to search whether the corresponding `fuse_entry_param` entry exists. If a valid entry is found, the reference count (`nlookup`) is incremented and a reply is sent to the FUSE driver.

Similarly, replies from user space daemon for other metadata operations, such as `getattr`, `getxattr`, and `readlink` could be cached using maps and served in the kernel by respective extensions (Table 4). Network FUSE file systems, such as SshFS [39] and Gluster [40] already perform aggressive metadata caching and batching at client to reduce the number of remote calls to the server. SshFS [39], for example, implements its own directory, attribute, and symlink caches. With EXTFUSE, such caches could be implemented in the kernel for further performance gains.

Invalidation. While caching metadata in the kernel reduces the number of context switches to user space, developers must also carefully invalidate replies, as necessary. For example, when a file (or dir) is deleted or renamed, the corresponding cached `lookup` replies must be invalidated. Invalidations can be performed in user space by the relevant request handlers or in the kernel by installing their extensions before new changes are made. However, the former case may introduce race conditions and produce incorrect results because all requests to user space daemon are queued up by the FUSE driver, whereas requests to the extensions are not. Cached `lookup` replies can be invalidated in extensions for `unlink`, `rmdir`, and `rename` operations. Similarly, when attributes or permissions on a file change, cached `getattr` replies can be invalidated in `setattr` extension. Our design ensures race-free invalidation by executing the extensions before forwarding requests to user space daemon where the changes may be made.

Advantages over VFS caching. As previously mentioned, recent optimizations added to FUSE framework leverage VFS caches to reduce user-kernel context switching. For instance, by specifying non-zero `entry_valid` and `attr_valid` timeout values, dentries and inodes cached by the VFS from previous `lookup` operations could be utilized to serve subsequent `lookup` and `getattr` requests, respectively. However, the VFS offers no control to the user file system over the cached data. For example, if the file system is mounted without the `default_permissions` parameter, VFS caching of inodes introduces a security bug [21]. This is because the cached permissions are only checked for first accessing user. In contrast, with EXTFUSE, developers can define their own metadata caches and install custom code to manage them. For instance, extensions can perform `uid`-based access permission checks before serving requests from the caches to obviate the aforementioned security issue (Figure 10).

Additionally, unlike VFS caches that are only reactive, EXTFUSE enables proactive caching. For example, since a `readdir` request is expected after an `opendir` call, the user-space daemon could proactively cache directory entries in the

Metadata	Map Key	Map Value	Caching Operations	Serving Extensions	Invalidation Operations
Inode	<nodeID, name>	fuse_entry_param	`lookup, create, mkdir, mknod`	*lookup*	`unlink, rmdir, rename`
Attrs	<nodeID>	fuse_attr_out	`getattr, lookup`	*getattr*	`setattr, unlink, rmdir`
Symlink	<nodeID>	link path	`symlink, readlink`	*readlink*	`unlink`
Dentry	<nodeID>	fuse_dirent	`opendir, readdir`	*readdir*	`releasedir, unlink, rmdir, rename`
XAttrs	<nodeID, label>	xattr value	`open, getxattr, listxattr`	*getattr, listxattr*	`close, setxattr, removexattr`

Table 4: Metadata can be cached in the kernel using eBPF maps by the user-space daemon and served by kernel extensions.

kernel by inserting them in a BPF map while serving `opendir` requests to reduce transitions to user space. Alternatively, similar to read-ahead optimization, proactive caching of subsequent directory entries could be performed during the first `readdir` call to the user-space daemon. Memory occupied by cached entries could then be freed by the `releasedir` handler in user space that deletes them from the map. Similarly, security labels on a file could be cached during the `open` call to user space and served in the kernel by `getxattr` extensions. Nonetheless, since eBPF maps consume kernel memory, developers must carefully manage caches and limit the number of map entries to keep memory usage under check.

5.2 Passthrough I/O for stacking functionality

Many user file systems are stackable with a thin layer of functionality that does not require complex processing in the user-space. For example, LoggedFS [16] filters requests that must be logged, logs them as needed, and then simply forwards them to the lower file system. User-space union file systems, such as MergerFS [20] determine the backend host file in `open` and redirects I/O requests to it. BindFS [35] mirrors another mount point with custom permissions checks. Android sdcard daemon performs access permission checks and emulates the case-insensitive behavior of FAT only in metadata operations (e.g., `lookup`, `open`, etc.), but forwards data I/O requests directly to the lower file system. For such simple cases, the FUSE API proves to be too low-level and incurs unnecessarily high overhead due to context switching.

With ExtFUSE, `read/write` I/O requests can take the fast path and directly be forwarded to the lower (host) file system without incurring any context-switching if the complex slow-path user-space logic is not needed. Figure 4 shows how the user-space daemon can install the lower file descriptor in `InodeMap` while handling `open()` system call for notifying the ExtFUSE driver to store a reference to the lower inode kernel object. With the `custom_filtering_logic(path)` condition, this can be done selectively; for example, if access permission checks pass in Android sdcard daemon. Similarly, BindFS and MergerFS can adopt ExtFUSE to avail passthrough optimization. The `read/write` kernel extensions can check in `InodeMap` to detect whether the target file is setup for passthrough access. If found, ExtFUSE driver can be instructed with a special return code to directly forward the I/O request to the lower file system with the corresponding lower inode object as parameter. Figure 5 shows a template `read` extension. Kernel extensions can include additional logic or checks before returning. For instance, LoggedFS `read/write` extensions can filter and log operations, as needed §6.2.

```
1  void handle_open(fuse_req_t req, fuse_ino_t ino,
2          const struct fuse_open_in *in) {
3      /* file represented by @ino inode num */
4      struct fuse_open_out out; char path[PATH_MAX];
5      int len, fd = open_file(ino, in->flags, path, &out);
6      if (fd > 0 && custom_filtering_logic(path)) {
7  +       /* install fd in inode map for pasthru */
8  +       imap_key_t key = out->fh;
9  +       imap_val_t val = fd; /* lower fd */
10 +       extfuse_insert_imap(&key, &val);
11 }   }
```

Figure 4: FUSE daemon `open` handler in user space. With ExtFUSE, lines 7-9 (+) enable passthrough access on the file.

```
1 int read_extension(extfuse_req_t req, fuse_ino_t ino,
2         const struct fuse_read_in *in) {
3     /* lookup in inode map, passthrough if exists */
4     imap_key_t key = in->fh;
5     if (!extfuse_lookup_imap(&key)) return UPCALL;
6     /* EXAMPLE: LoggedFS log operation */
7     log_op(req, ino, FUSE_READ, in, sizeof(*in));
8     return PASSTHRU; /* forward req to lower FS */
9 }
```

Figure 5: The ExtFUSE `read` kernel extension returns `PASSTHRU` to forward request directly to the lower file system. Custom thin functionality could further be pushed in the kernel; LoggedFS logging function is shown as an example (see Figure 11).

6 Evaluation

To evaluate ExtFUSE, we answer the following questions:

- **Baseline Performance.** How does an ExtFUSE implementation of a file system perform when compared to its in-kernel and FUSE implementations? (§6.1)
- **Use cases.** What kind of existing FUSE file systems can benefit from ExtFUSE and what performance improvements can they expect? (§6.2)

6.1 Performance

To measure the baseline performance of ExtFUSE, we adopted the simple no-ops (null) stackable FUSE file system called Stackfs [50]. This user-space daemon serves all requests by forwarding them to the host (lower) file system. It includes all recent FUSE optimizations (Table 5). We evaluate Stackfs under all possible ExtFUSE configs listed in Table 5. Each config represents a particular level of performance that could potentially be achieved, for example, by caching metadata in the kernel or directly passing `read/write` requests through the host file system for stacking functionality. To put our results in context, we compare our results with EXT4 and

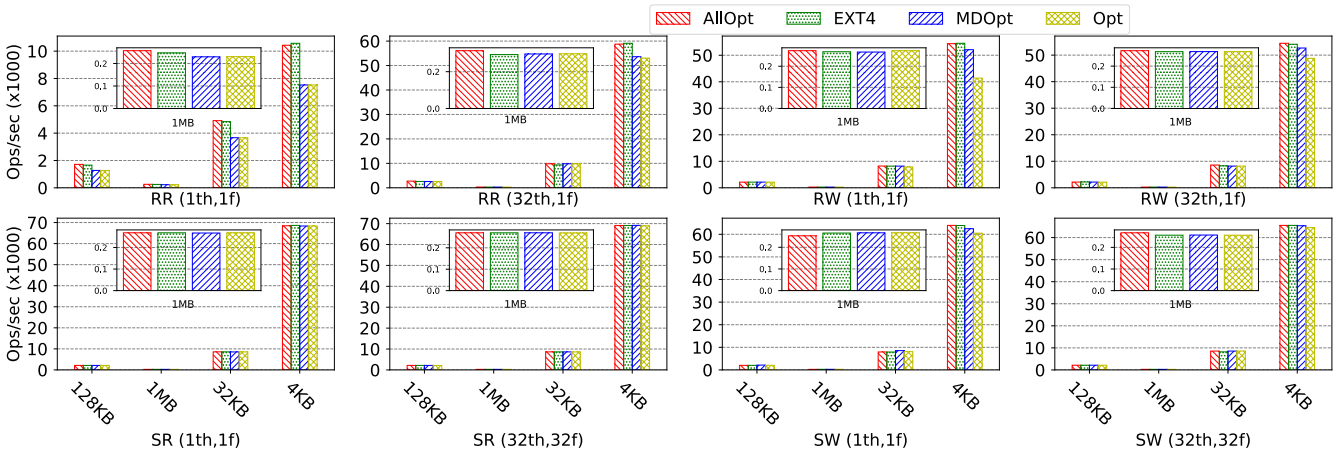

Figure 6: Throughput(ops/sec) for EXT4 and FUSE/EXTFUSE Stackfs (w/ xattr) file systems under different configs (Table 5) as measured by Random Read(RR)/Write(RW), Sequential Read(SR)/Write(SW) Filebench [48] data micro-workloads with IO Sizes between 4KB-1MB and settings *N*th: *N* threads, *N*f: *N* files. We use the same workloads as in [50].

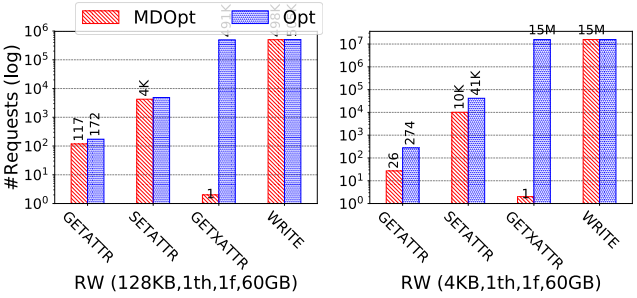

Figure 7: Number of file system request received by the daemon in FUSE/EXTFUSE Stackfs (w/ xattr) under workloads in Figure 6. Only a few relevant request types are shown.

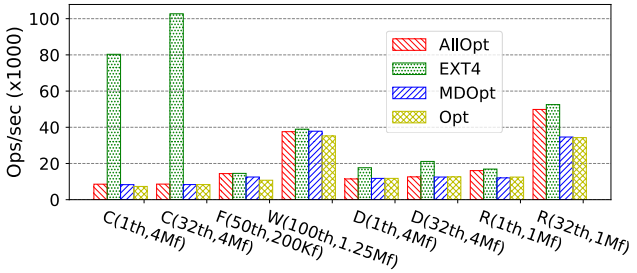

Figure 8: Throughput(Ops/sec) for EXT4 and FUSE/EXTFUSE Stackfs (w/ xattr) under different configs (Table 5) as measured by Filebench [48] Creation(C), Deletion(D), Reading(R) metadata micro-workloads on 4KB files and FileServer(F), WebServer(W) macro-workloads with settings *N*th:*N* threads, *N*f:*N* files.

the optimized FUSE implementation of Stackfs (`Opt`).

Testbed. We use the same experiments and settings as in [50]. Specifically, we used EXT4 because of its popularity as the host file system and ran benchmarks to evaluate. However, since FUSE performance problems were reported to be more prominent with a faster storage medium, we only carry out our experiments with SSDs. We used a Samsung 850 EVO 250GB SSD installed on an Asus machine with Intel Quad-Core i5-3550 3.3 GHz and 16GB RAM, running Ubuntu 16.04.3. Further, to minimize any variability, we formatted the SSD before each experiment and disabled EXT4 lazy inode initialization. To evaluate file systems that implement `xattr` operations for handling security labels (e.g., in Android), our implementation of `Opt` supports xattrs, and thus differs from the implementation in [50].

Config	File System	Optimizations
`Opt` [50]	FUSE	128K Writes, Splice, WBCache, MltThrd
`MDOpt`	EXTFUSE	`Opt` + Caches `lookup`, `attrs`, `xattrs`
`AllOpt`	EXTFUSE	`MDOpt` + Pass `R/W` reqs through host FS

Table 5: Different Stackfs configs evaluated.

Workloads. Our workload consists of Filebench [48] micro and synthetic macro benchmarks to test each config with metadata- and data-heavy operations across a wide range of I/O sizes and parallelism settings. We measure the low-level throughput (ops/sec). Our macro-benchmarks consist of a synthetic file server and web server.

Micro Results. Figure 6 shows the results of micro workload under different configs listed in Table 5.

Reads. Due to the default 128KB read-ahead feature of FUSE, the sequential read throughput on a single file with a single thread for all I/O sizes and under all Stackfs configs remained the same. Multi-threading improved for the sequential read benchmark with 32 threads and 32 files. Only one request was generated per thread for `lookup` and `getattr` operations. Hence, metadata caching in `MDOpt` was not effective. Since FUSE `Opt` performance is already at par with EXT4, the passthrough feature in `AllOpt` was not utilized.

Unlike sequential reads, small random reads could not take advantage of the read-ahead feature of FUSE. Additionally, 4KB reads are not spliced and incur data copying across user-kernel boundary. With 32 threads operating on a single file, the throughput improves due to multi-threading in `Opt`. However, degradation is observed with 4KB reads. `AllOpt` passes all reads through EXT4, and hence offers near-native throughput. In some cases, the performance was slightly better than

EXT4. We believe that this minor improvement is due to double caching at the VFS layer. Due to a single request per thread for metadata operations, no improvement was seen with ExtFUSE metadata caching.

Writes. During sequential writes, the 128K big writes and writeback caching in `Opt` allow the FUSE driver to batch small writes (up to 128KB) together in the page cache to offer a higher throughput. However, random writes are not batched. As a result, more `write` requests are delivered to user space, which negatively affects the throughput. Multiple threads on a single file perform better for requests bigger than 4KB as they are spliced. With ExtFUSE `AllOpt`, all writes are passed through the EXT4 file system to offer improved performance.

Write throughput degrades severely for FUSE file systems that support extended attributes because the VFS issues a `getxattr` request before every `write`. Small I/O requests perform worse as they require more `write`, which generate more `getxattr` requests. `Opt` random writes generated 30x fewer `getxattr` requests for 128KB compared to 4KB writes, resulting in a 23% decrease in the throughput of 4KB writes.

In contrast, `MDOpt` caches the `getxattr` reply in the kernel upon the first call, and serves subsequent `getxattr` requests without incurring further transitions to user space. Figure 7 compares the number of requests received by the user-space daemon in `Opt` and `MDOpt`. Caching replies reduced the overhead for 4KB workload to less than 5%. Similar behavior was observed with both sequential writes and random writes.

Macro Results. Figure 8 shows the results of macro-workloads and synthetic server workloads emulated using Filebench under various configs. Neither of the ExtFUSE configs offer improvements over FUSE `Opt` under creation and deletion workloads as these metadata-heavy workloads created and deleted a number of files, respectively. This is because no metadata caching could be utilized by `MDOpt`. Similarly, no passthrough writes were utilized with `AllOpt` since 4KB files were created and closed in user space. In contrast, the File and Web server workloads under ExtFUSE utilized both metadata caching and passthrough access features and improved performance. We saw a 47%, 89%, and 100% drop in `lookup`, `getattr`, and `getattr` requests to user space under `MDOpt`, respectively, when configured to cache up to 64K for each type of request. `AllOpt` further enabled passthrough `read/write` requests to offer near native throughput for both macro reads and server workloads.

Real Workload We also evaluated ExtFUSE with two real workloads, namely kernel decompression and compilation of 4.18.0 Linux kernel. We created three separate caches for hosting `lookup`, `getattr`, and `getxattr` replies. Each cache could host up to 64K entries, resulting in allocation of up to a total of 50MB memory when fully populated.

The kernel compilation `make tinyconfig; make -j4` experiment on our test machine (see §6) reported a 5.2% drop in compilation time, from 39.74 secs under FUSE `Opt` to 37.68 secs with ExtFUSE `MDOpt`, compared to 30.91 secs with EXT4. This was due to over 75%, 99%, and 100% decrease in `lookup`, `getattr`, and `getxattr` requests to user space, respectively (Figure 9). `getxattr` replies were proactively cached while handling `open` requests; thus, no transitions to user space were observed for serving xattr requests. With ExtFUSE `AllOpt`, the compilation time further dropped to 33.64 secs because of 100% reduction in `read` and `write` requests to user space.

In contrast, the kernel decompression `tar xf` experiment reported a 6.35% drop in the completion time, from 11.02 secs under FUSE `Opt` to 10.32 secs with ExtFUSE `MDOpt`, compared to 5.27 secs with EXT4. With ExtFUSE `AllOpt`, the decompression time further dropped to 8.67 secs due to 100% reduction in `read` and `write` requests to user space, as shown in Figure 9. Nevertheless, reducing the number of cached entries for metadata requests to 4K resulted in a decompression time of 10.87 secs (25.3% increase) due to 3,555 more `getattr` requests to user space. This suggests that developers must efficiently manage caches.

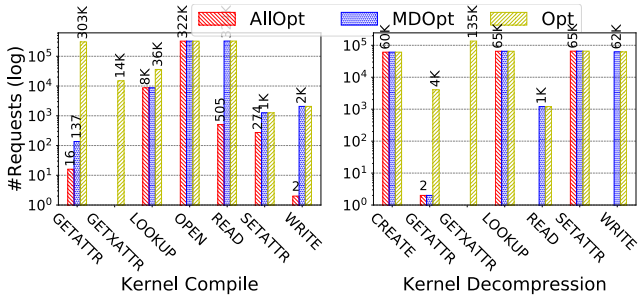

Figure 9: Linux kernel 4.18.0 untar (decompress) and compilation time taken with StackFS under FUSE and ExtFUSE settings. Number of metadata and I/O requests are reduced with ExtFUSE.

6.2 Use cases

We ported four real-world stackable FUSE file systems, namely LoggedFS, Android sdcard daemon, MergerFS, and BindFS to ExtFUSE and enabled both metadata caching §5.1 and passthrough I/O §5.2 optimizations.

File System	Functionality	Ext Loc
StackFS [50]	No-ops File System	664
BindFS [35]	Mirroring File System	792
Android sdcard [24]	Perm checks & FAT Emu	928
MergerFS [20]	Union File System	686
LoggedFS [16]	Logging File System	748

Table 6: Lines of code (Loc) of kernel extensions required to adopt ExtFUSE for existing FUSE file systems. We added support for metadata caching as well as R/W passthrough.

As ExtFUSE allows file systems to retain their existing FUSE daemon code as the default slow path, adopting ExtFUSE for real-world file systems is easy. On average, we made less than 100 lines of changes to the existing FUSE code to invoke ExtFUSE helper library functions for manipulating kernel extensions, including maps. We added ker-

nel extensions to support metadata caching as well as I/O passthrough. Overall, it required fewer than 1000 lines of new code in the kernel Table 6. We now present detailed evaluation of Android sdcard daemon and LoggedFS to present an idea on expected performance improvements.

Android sdcard daemon. Starting version 3.0, Android introduced the support for FUSE to allow a large part of internal storage (e.g., `/data/media`) to be mounted as external FUSE-managed storage (called `/sdcard`). Being large in size, `/sdcard` hosts user data, such as videos and photos as well as any auxiliary Opaque Binary Blobs (OBB) needed by Android apps. The FUSE daemon enforces permission checks in metadata operations (e.g.,`lookup`, etc.) on files under `/sdcard` to enable multi-user support and emulates case-insensitive FAT functionality on the host (e.g., EXT4) file system. OBB files are compressed archives and typically used by games to host multiple small binaries (e.g. shade rs, textures) and multimedia objects (e.g. images, etc.).

However, FUSE incurs high runtime performance overhead. For instance, accessing OBB archive content through the FUSE layer leads to high launch latency and CPU utilization for gaming apps. Therefore, Android version 7.0 replaced sdcard daemon with with an in-kernel file system called `SDCardFS` [24] to manage external storage. It is a wrapper (thin) stackable file system based on WrapFS [54] that enforces permission checks and performs FAT emulation in the kernel. As such, it imposes little to no overhead compared to its user-space implementation. Nevertheless, it introduces security risks and maintenance costs [12].

We ported Android sdcard FUSE daemon to ExtFUSE framework. First, we leverage eBPF kernel helper functions to push metadata checks into the kernel. For example, we embed access permission check (Figure 10) in `lookup` kernel extension to validate access before serving `lookup` replies from the cache (Figure 3). Similar permission checks are performed in the kernel to validate accesses to files under `/sdcard` before serving cached `getattr` requests . We also enabled passthrough on `read/write` using `InodeMap`.

We evaluated its performance on a 1GB RAM HiKey620 board [1] with two popular game apps containing OBB files of different sizes. Our results show that under `AllOpt` passthrough mode the app launch latency and the corresponding peak CPU consumption reduces by over 90% and 80%, respectively. Furthermore, we found that the larger the OBB file, the more penalty is incurred by FUSE due to many more small files in the OBB archive.

LoggedFS is a FUSE-based stackable user-space file system. It logs every file system operation for monitoring purposes. By default it writes to syslog buffer and logs all operations (e.g., open, read, etc.). However, it can be configured to write to a file or log selectively. Despite being a simple file system, it has a very important use case. Unlike existing monitoring mechanisms (e.g., Inotify [31]) that suffer from a host of limitations [10], LoggedFS can reliably post all file system

App Stats		CPU (%)		Latency (ms)	
Name	**OBB Size**	**D**	**P**	**D**	**P**
Disney Palace Pets 5.1	374MB	20	2.9	2235	1766
Dead Effect 4	1.1GB	20.5	3.2	8895	4579

Table 7: App launch latency and peak CPU consumption of `sdcard` daemon under default (D), and passthrough (P) settings on Android for two popular games. In passthrough mode, the FUSE driver never forwards `read/write` requests to user space, but always passes them through the host (EXT4) file system. See Table 5 for config details.

```
bool check_caller_access_to_name(int64_t key, const char *name) {
    /* define a shmap for hosting permissions */
    int *val = extfuse_lookup_shmap(&key);
    /* Always block security-sensitive files at root */
    if (!val || *val == PERM_ROOT) return false;
    /* special reserved files */
    if (!strncasecmp(name, "autorun.inf", 11) ||
        !strncasecmp(name, ".android_secure", 15) ||
        !strncasecmp(name, "android_secure", 14))
        return false;
    return true;
}
```

Figure 10: Android `sdcard` permission checks ExtFUSE code.

events. Various apps, such as file system indexers, backup tools, Cloud storage clients such as Dropbox, integrity checkers, and antivirus software subscribe to file system events for efficiently tracking modifications to files.

We ported LoggedFS to ExtFUSE framework. Figure 11 shows the common logging code that is called from various extensions, which serve requests in the kernel (e.g., `read` extension Figure 5). To evaluate its performance, we ran the FileServer macro benchmark with synthetic a workload of 200,000 files and 50 threads from Filebench suite. We found over 9% improvement in throughput under `MDOpt` compared to FUSE `Opt` due to 53%, 99%, and 100% fewer `lookup`, `getattr`, and `getxattr` requests to user space, respectively. Figure 12 shows the results. `AllOpt` reported an additional 20% improvement by directly forwarding all `read/write` requests to the host file system, offering near-native throughput.

```
void log_op(extfuse_req_t req, fuse_ino_t ino,
        int op, const void *arg, int arglen) {
    struct data { /* log record */
        u32 op; u32 pid; u64 ts; u64 ino; char data[MAXLEN];};
    /* example filter: only whitelisted UIDs in map */
    u16 uid = bpf_get_current_uid_gid();
    if (!extfuse_lookup_shmap(uid_wlist, &uid)) return;
    /* log opcode, timestamp(ns) and requesting process */
    data.opcode = op; data.ts  = bpf_ktime_get_ns();
    data.pid = bpf_get_current_pid_tgid(); data.ino = ino;
    memcpy(data.data, arg, arglen);
    /* submit to per-cpu mmap'd ring buffer */
    u32 key = bpf_get_smp_processor_id();
    bpf_perf_event_output(req, &buf, &key, &data, sizeof(data));
}
```

Figure 11: LoggedFS kernel extension that logs requests.

7 Discussion

Future use cases. Given negligible overhead of ExtFUSE and direct passthrough access to the host file system for stacking incremental functionality, multiple app-defined "thin" file system functions (e.g., security checks, logging, etc.) can be

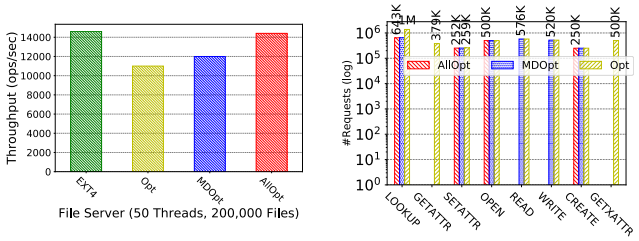

Figure 12: LoggedFS performance measured by Filebench FileServer benchmark under EXT4, FUSE, and EXTFUSE. Fewer metadata and I/O requests were delivered to user space with EXTFUSE.

stacked with low overhead, which otherwise would have been very expensive in user space with FUSE.

Safety. EXTFUSE sandboxes untrusted user extensions to guarantee safety. For example, the eBPF runtime allows access to only a few, simple non-blocking kernel helper functions. Map data structures are of fixed size. Extensions are not allowed to allocate memory or directly perform any I/O operations. Even so, EXTFUSE offers significant performance boost across a number of use cases §6.2 by offloading simple logic in the kernel. Nevertheless, with EXTFUSE, user file systems can retain their existing slow-path logic for performing complex operations, such as encryption in user space. Future work can extend the EXTFUSE framework to take advantage of existing generic in-kernel services such as VFS encryption and compression APIs to even serve requests that require such complex operations entirely in the kernel.

8 Related Work

Here, we compare our work with related existing research.

User File System Frameworks. There exists a number of frameworks to develop user file systems. A number of user file systems have been implemented using NFS loopback servers [19]. UserFS [14] exports generic VFS-like file system requests to the user space through a file descriptor. Arla [53] is an AFS client system that lets apps implement a file system by sending messages through a device file interface `/dev/xfs0`. Coda file system [42] exported a similar interface through `/dev/cfs0`. NetBSD provides Pass-to-Userspace Framework FileSystem (PUFFS). Mazières et al. proposed a C++ toolkit that exposes a NFS-like interface for allowing file systems to be implemented in user space [33]. UFO [3] is a global file system implemented in user space by introducing a specialized layer between the apps and the OS that intercepts file system calls.

Extensible Systems. Past works have explored the idea of letting apps extend system services at runtime to meet their performance and functionality needs. SPIN [6] and VINO [43] allow apps to safely insert kernel extensions. SPIN uses a type-safe language runtime, whereas VINO uses software fault isolation to provide safety. ExoKernel [13] is another OS design that lets apps define their functionality. Systems such as ASHs [17, 51] and Plexus [15] introduced the concept of network stack extension handlers inserted into the kernel. SLIC [18] extends services in monolithic OS using interposition to enable incremental functionality and composition. SLIC assumes that extensions are trusted. EXTFUSE is a framework that allows user file systems to add "thin" extensions in the kernel that serve as specialized interposition layers to support both in-kernel and user space processing to co-exist in monolithic OSes.

eBPF. EXTFUSE is not the first system to use eBPF for safe extensibility. eXpress DataPath (XDP) [27] allows apps to insert eBPF hooks in the kernel for faster packet processing and filtering. Amit et al. proposed Hyperupcalls [4] as eBPF helper functions for guest VMs that are executed by the hypervisor. More recently, SandFS [7] uses eBPF to provide an extensible file system sandboxing framework. Like EXTFUSE, it also allows unprivileged apps to insert custom security checks into the kernel.

FUSE. File System Translator (FiST) [55] is a tool for simplifying the development of stackable file system. It provides *boilerplate* template code and allows developers to only implement the core functionality of the file system. FiST does not offer safety and reliability as offered by user space file system implementation. Additionally, it requires learning a slightly simplified file system language that describes the operation of the stackable file system. Furthermore, it only applies to stackable file systems.

Narayan et al. [34] combined in-kernel stackable FiST driver with FUSE to offload data from I/O requests to user space to apply complex functionality logic and pass processed results to the lower file system. Their approach is only applicable to stackable file systems. They further rely on static per-file policies based on extended attributes labels to enable or disable certain functionality. In contrast, EXTFUSE downloads and safely executes thin extensions from user file systems in the kernel that encapsulate their rich and specialized logic to serve requests in the kernel and skip unnecessary user-kernel switching.

9 Conclusion

We propose the idea of extensible user file systems and present the design and architecture of EXTFUSE, an extension framework for FUSE file system. EXTFUSE allows FUSE file systems to define "thin" extensions along with auxiliary data structures for specialized handling of low-level requests in the kernel while retaining their existing complex logic in user space. EXTFUSE provides familiar FUSE-like APIs and abstractions for easy adoption. We demonstrate its practical usefulness, suitability for adoption, and performance benefits by porting and evaluating existing FUSE implementations.

10 Acknowledgments

We thank our shepherd, Dan Williams, and all anonymous reviewers for their feedback, which improved the content of this paper. This work was funded in part by NSF CPS program Award #1446801, and a gift from Microsoft Corp.

References

[1] 96boards. Hikey (lemaker) development boards, May 2019.

[2] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. pages 93–112, 1986.

[3] Albert D Alexandrov, Maximilian Ibel, Klaus E Schauser, and Chris J Scheiman. Extending the operating system at the user level: the Ufo global file system. In *Proceedings of the 1997 USENIX Annual Technical Conference (ATC)*, pages 6–6, Anaheim, California, January 1997.

[4] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, pages 97–112, Boston, MA, July 2018.

[5] Starr Andersen and Vincent Abella. Changes to Functionality in Windows XP Service Pack 2, Part 3: Memory Protection Technologies, 2004. https://technet.microsoft.com/en-us/library/bb457155.aspx.

[6] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, December 1995.

[7] Ashish Bijlani and Umakishore Ramachandran. A lightweight and fine-grained file system sandboxing framework. In *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys)*, Jeju Island, South Korea, August 2018.

[8] Ceph. Ceph kernel client, April 2018. https://github.com/ceph/ceph-client.

[9] Open ZFS Community. ZFS on Linux. https://zfsonlinux.org, April 2018.

[10] J. Corbet. Superblock watch for fsnotify, April 2017.

[11] Brian Cornell, Peter A. Dinda, and Fabián E. Bustamante. Wayback: A user-level versioning file system for linux. In *Proceedings of the 2004 USENIX Annual Technical Conference (ATC)*, pages 27–27, Boston, MA, June–July 2004.

[12] Exploit Database. Android - sdcardfs changes current->fs without proper locking, 2019.

[13] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, December 1995.

[14] Jeremy Fitzhardinge. Userfs, March 2018. http://www.goop.org/~jeremy/userfs/.

[15] Marc E. Fiuczynski and Briyan N. Bershad. An Extensible Protocol Architecture for Application-Specific Networking. In *Proceedings of the 1996 USENIX Annual Technical Conference (ATC)*, San Diego, CA, January 1996.

[16] R. Flament. LoggedFS - Filesystem monitoring with Fuse, March 2018. https://rflament.github.io/loggedfs/.

[17] Gregory R. Ganger, Dawson R. Engler, M. Frans Kaashoek, Hector M. Briceño, Russell Hunt, and Thomas Pinckney. Fast and Flexible Application-level Networking on Exokernel Systems. *ACM Transactions on Computer Systems (TOCS)*, 20(1):49–83, 2002.

[18] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. Slic: An extensibility system for commodity operating systems. In *Proceedings of the 1998 USENIX Annual Technical Conference (ATC)*, New Orleans, Louisiana, June 1998.

[19] David K Gifford, Pierre Jouvelot, Mark A Sheldon, et al. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 16–25, Pacific Grove, CA, October 1991.

[20] A Featureful Union Filesystem, March 2018. https://github.com/trapexit/mergerfs.

[21] LibFuse | GitHub. Without 'default_permissions', cached permissions are only checked on first access, 2018. https://github.com/libfuse/libfuse/issues/15.

[22] Gluster. libgfapi, April 2018. http://staged-gluster-docs.readthedocs.io/en/release3.7.0beta1/Features/libgfapi/.

[23] Gnu hurd, April 2018. www.gnu.org/software/hurd/hurd.html.

[24] Storage | Android Open Source Project, September 2018. https://source.android.com/devices/storage/.

[25] John H Hartman and John K Ousterhout. Performance measurements of a multiprocessor sprite kernel. In *Proceedings of the Summer 1990 USENIX Annual Technical Conference (ATC)*, pages 279–288, Anaheim, CA, 1990.

[26] eBPF: extended Berkley Packet Filter, 2017. https://www.iovisor.org/technology/ebpf.

[27] IOVisor. Xdp - io visor project, May 2019.

[28] Shun Ishiguro, Jun Murakami, Yoshihiro Oyama, and Osamu Tatebe. Optimizing local file accesses for fuse-based distributed storage. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 760–765, 2012.

[29] Antti Kantee. puffs-pass-to-userspace framework file system. In *Proceedings of the Asian BSD Conference (AsiaBSDCon)*, Tokyo, Japan, March 2007.

[30] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, pages 175–188, Asheville, NC, December 1993.

[31] Robert Love. Kernel korner: Intro to inotify. *Linux Journal*, 2005:8, 2005.

[32] Ali José Mashtizadeh, Andrea Bittau, Yifeng Frank Huang, and David Mazières. Replication, history, and grafting in the ori file system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, November 2013.

[33] David Mazières. A Toolkit for User-Level File Systems. In *Proceedings of the 2001 USENIX Annual Technical Conference (ATC)*, pages 261–274, June 2001.

[34] S. Narayan, R. K. Mehta, and J. A. Chandy. User space storage system stack modules with file level control. In *Proceedings of the Linux Symposium*, pages 189–196, Ottawa, Canada, July 2010.

[35] Martin Pärtel. bindfs, 2018. https://bindfs.org.

[36] PaX Team. PaX address space layout randomization (ASLR), 2003. https://pax.grsecurity.net/docs/aslr.txt.

[37] Rogério Pontes, Dorian Burihabwa, Francisco Maia, João Paulo, Valerio Schiavoni, Pascal Felber, Hugues Mercier, and Rui Oliveira. Safefs: A modular architecture for secure user-space file systems: One fuse to rule them all. In *Proceedings of the 10th ACM International on Systems and Storage Conference*, pages 9:1–9:12, Haifa, Israel, May 2017.

[38] Nikolaus Rath. List of fuse file systems, 2011. https://github.com/libfuse/libfuse/wiki/Filesystems.

[39] Nicholas Rauth. A network filesystem client to connect to SSH servers, April 2018. https://github.com/libfuse/sshfs.

[40] Gluster, April 2018. http://gluster.org.

[41] Kai Ren and Garth Gibson. Tablefs: Enhancing metadata efficiency in the local file system. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 145–156, San Jose, CA, June 2013.

[42] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[43] Margo Seltzer, Yasuhiro Endo, Christopher Small, and Keith A Smith. An introduction to the architecture of the vino kernel. Technical report, Technical Report 34-94, Harvard Computer Center for Research in Computing Technology, October 1994.

[44] Helgi Sigurbjarnarson, Petur O. Ragnarsson, Juncheng Yang, Ymir Vigfusson, and Mahesh Balakrishnan. Enabling space elasticity in storage systems. In *Proceedings of the 9th ACM International on Systems and Storage Conference*, pages 6:1–6:11, Haifa, Israel, June 2016.

[45] David C Steere, James J Kistler, and Mahadev Satyanarayanan. Efficient user-level file cache management on the sun vnode interface. In *Proceedings of the Summer 1990 USENIX Annual Technical Conference (ATC)*, Anaheim, CA, 1990.

[46] Swaminathan Sundararaman, Laxman Visampalli, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Refuse to Crash with Re-FUSE. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, Salzburg, Austria, April 2011.

[47] M. Szeredi and N.Rauth. Fuse - filesystems in userspace, 2018. https://github.com/libfuse/libfuse.

[48] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine*, 41(1):6–12, March 2016.

[49] Ungureanu, Cristian and Atkin, Benjamin and Aranya, Akshat and Gokhale, Salil and Rago, Stephen and Całkowski, Grzegorz and Dubnicki, Cezary and Bohra, Aniruddha. HydraFS: A High-throughput File System for the HYDRAstor Content-addressable Storage System. In *10th USENIX Conference on File and Storage Technologies (FAST) (FAST 10)*, pages 17–17, San Jose, California, February 2010.

[50] 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) (FAST 17)*, Santa Clara, CA, February 2017.

[51] Deborah A. Wallach, Dawson R. Engler, and M. Frans Kaashoek. ASHs: Application-Specific Handlers for High-Performance Messaging. In *Proceedings of the 7th ACM SIGCOMM*, Palo Alto, CA, August 1996.

[52] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–320, Seattle, WA, November 2006.

[53] Assar Westerlund and Johan Danielsson. Arla: a free afs client. In *Proceedings of the 1998 USENIX Annual Technical Conference (ATC)*, pages 32–32, New Orleans, Louisiana, June 1998.

[54] E. Zadok, I. Bădulescu, and A. Shender. Extending File Systems Using Stackable Templates". In *Proceedings of the 1999 USENIX Annual Technical Conference (ATC)*, pages 57–70, June 1999.

[55] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference (ATC)*, June 2000.

[56] Erez Zadok, Sean Callanan, Abhishek Rai, Gopalan Sivathanu, and Avishay Traeger. Efficient and safe execution of user-level code in the kernel. In *Parallel and Distributed Processing Symposium 19th IEEE International*, pages 8–8, 2005.

[57] ZFS-FUSE, April 2018. https://github.com/zfs-fuse/zfs-fuse.