Trust in a Principled Filesystem

Kevin Walsh and Fred B. Schneider

Abstract—Users typically have little reason to trust that systems will protect user data against unauthorized access. A prototype filesystem stack was implemented with a structure designed to provide such assurance by preventing filesystem administrators and most filesystem code from violating user data security goals. The design instantiates a number of well-known security principles that have been proposed for the design of trustworthy systems. Benchmark results indicate the associated performance impact can be low.

Index Terms—Complete Mediation; Least Privilege; Mutual Suspicion; Trusted Computing Bases.

1 Introduction

Users implicitly trust the systems on which they store data to respect certain data integrity and confidentiality properties. On a multi-user server, each user would expect the kernel and filesystem stack not to leak the data to other users, for example, and users might expect that system administrators will not access or modify user data without explicit permission. Users of shared network storage systems have similar expectations. Likewise, applications running on a multi-tenant cloud platform typically expect the platform to safeguard data against unauthorized access by competing applications, both for local and for distributed cloud storage systems.

A variety of techniques have been employed to help justify or eliminate the need for such trust. If data is stored on disk in encrypted form before writing to local disks or remote storage, for example, then an operator having access only to read disks or observe network data need not be trusted by the user. Or, we might rely on the capabilities of a TPM secure co-processor to allow users to audit code and configuration parameters for a remote platform before deciding what level of trust is appropriate for that platform. Still, in both scenarios, a user's trust is justified only to the extent that all security-relevant code and all accompanying security-relevant configuration is known to and trusted by the user, including all code that could directly or indirectly access user data, whether stored on disk or resident in memory.

- *K. Walsh is with the Department of Mathematics and Computer Science, College of the Holy Cross, Worcester, MA, 01610.*
- *F. B. Schneider is with the Department of Computer Science, Cornell University, Ithaca, NY, 14853.*

Supported in part by NICECAP cooperative agreement FA8750-07-2-0037 administered by AFRL, AFOSR grant F9550-06-0019, National Science Foundation grants 0430161, 0964409, and CCF-0424422 (TRUST), ONR grants N00014-01-1-0968 and N00014-09-1-0652, and grants from Microsoft. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

We implemented a prototype filesystem stack called PFS (Principled Filesystem) to explore whether restructuring the security-relevant code base could lead to higher user assurance about data confidentiality and integrity. PFS pervasively instantiates several principles that are discussed but rarely put into practice (due to performance concerns) in the design and implementation of trustworthy systems: Mutual Suspicion [1], Complete Mediation [2], Least Privilege [3], and Small Trusted Computing Bases [4]. An analysis of PFS performance thus gives insight into the impact these principles have in practice.

PFS exports a POSIX-like filesystem interface to applications. It is designed to run on α-Nexus, a variant of the Nexus [5] micro-kernel, and PFS both relies on features of that environment internally and exposes α-Nexus features to applications. We measured the performance of PFS using micro-benchmarks. Contrary to common wisdom, we found that several common operations could execute in PFS with performance on par with conventional Linux and α-Nexus implementations. And although we observed an increase in the amount of filesystem code overall in PFS compared to conventional implementations, the actual trusted computing base (TCB)—code whose compromise could result in a violation of our security goals—was greatly reduced in size.

1.1 Security Goals

Traditionally, filesystems enforce discretionary access control (DAC) policies that govern user access to files and directories. PFS enforces *pervasive* DAC, which governs access to every byte stored on disk, including boot sectors, partition tables, inode contents, and free block lists. This approach is summarized by a familiar security principle:

> **Complete Mediation:** Authorize, using an appropriate enforcement mechanism, every request to perform some action [2].

Each sequence of bytes stored by PFS on disk has an owner and an access control list (ACL), specified by

that owner or some principal explicitly authorized to act on the owner's behalf. Every request to access PFS data is checked against the ACL, whether the data is stored on disk or resides in the block cache. Further, PFS enforces DAC at a fine granularity. Thus an ACL might associate access rights not just with a user, but with a specific processes (identified by process ID), for example, or with a specific program (identified by hash of relevant code and configuration). In fact, components implementing PFS itself are considered principals and are subject to DAC policy enforcement.

1.2 Limitations

To simplify our presentation, we consider a single machine running PFS and hosting a set of mutually-distrusting application processes. PFS runs on standard Intel x86-compatible hardware with SATA disk buses. PFS does not encrypt data stored on disk or in memory, and we do not consider adversaries with physical access to the system's disks, memory, processors, or other hardware.

PFS trusts certain services provided by α-Nexus and trusts that α-Nexus will not itself make unauthorized accesses to user data. α-Nexus provides assurance of its own trustworthiness in part by relying on a TPM. PFS relies upon protection boundaries enforced by hardware and by the α-Nexus kernel. Such boundaries are relatively expensive and inflexible; software or language-based protection boundaries, such as those used in Singularity [6], would involve a different set of tradeoffs.

Our PFS implementation is a research prototype and does not support all POSIX filesystem features. For ease of implementation, PFS also currently supports only FAT32-formatted disk partitions. For these reasons, we do not address crash recovery here and we evaluate PFS under primarily read-only workloads. More modern filesystem formats could be supported by adapting available drivers.

2 Design and Implementation of PFS

PFS is organized into three main layers, augmented by several modules for cross-cutting concerns. A bottom layer comprises a set of *disk drivers*, to manage DMA transfers between system memory and disks. PFS disk drivers are based on the Linux libATA driver library. A middle layer comprises *filesystem drivers*, each implementing a standard file and directory view of one logical disk partition. Where necessary, we modified on-disk formats to support pervasive DAC, including for FAT32 file data and metadata and for legacy boot sectors and partition tables. A top layer, called the virtual filesystem (VFS), presents a unified namespace to applications and implements POSIX streams-oriented and mmap-oriented interfaces.

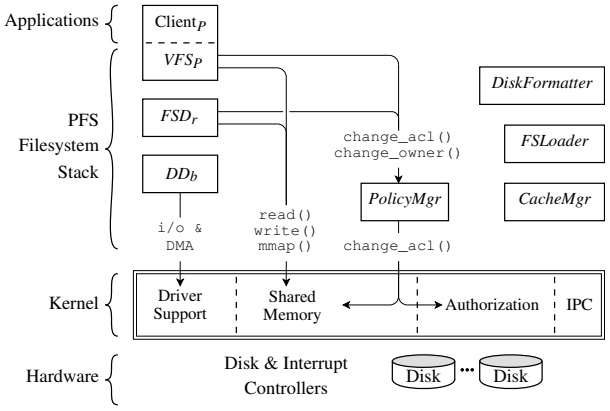

Fig. 1. Structure of the PFS implementation, showing major components and some of the interfaces between them. Boxes indicate hardware-enforced process and kernel isolation boundaries. Dashed lines indicate logical boundaries without isolation.

We intend PFS to be trustworthy, but we accept the reality that any implementation will have vulnerabilities. So our design attempts to blunt the impact that a successful attack on PFS might have—on PFS itself, on applications that use PFS, on the underlying operating system, or on applications and services unrelated to the filesystem. To mitigate against such risks, PFS executes outside of the operating system kernel. This exemplifies a well-known design principle:

> **Small Trusted Computing Bases:** Make the TCB for each security goal as small as possible, consistent with the tasks it must perform [4].

A smaller TCB should be easier to analyze; a larger TCB is more likely to have bugs, hence is more easily attacked. In contrast to PFS, conventional filesystem implementations locate the bulk of their code within the operating system kernel and thus become part of the TCB for every security goal, even goals unrelated to the filesystem.

To shrink the size of TCBs for enforcing pervasive DAC on behalf of each user—henceforth, *user TCBs*—we decompose PFS into more fine-grained components. The resulting structure, shown in Figure 1, includes: *DiskFormatter*, to execute code for creating and destroying disk partitions; *FSLoader*, to examine partition tables and load appropriate filesystem drivers; *CacheMgr*, to implement prefetching and eviction policies for cached data blocks; and *PolicyMgr*, to manage owner and ACL information; and several components for VFS, filesystem, and disk driver layers.

Decomposition alone does not reduce TCB size. Components must also be *isolated* so that critical functionality in one component is not affected by compromises in unrelated components. Two classic security principles provide guidance:

Mutual Suspicion: Prefer designs that reduce the

likelihood that any compromised component can cause the compromise of another [1].

Least Privilege: Grant each component the fewest privileges necessary to complete its task [3].

PFS exemplifies these principles by executing each filesystem component as a separate, isolated and unprivileged α-Nexus process, each holding only a narrow and specific set of privileges. $CacheMgr$, for example, is granted privileges sufficient for triggering loads into the cache and for evicting previously cached blocks, but it does not have privileges to access the contents of those blocks, nor does it have the ability to directly issue DMA requests. This removes $CacheMgr$ from user TCBs because a compromise of $CacheMgr$ does not result in unexpected accesses to user data.

Certain PFS components, such as $PolicyMgr$, are critical for pervasive DAC enforcement and thus are included in user TCBs. But an explicit design goal for PFS was to decompose the filesystem stack in a way that reduces the size and number of such components.

A design with finer-grained components may lead to smaller trusted computing bases, but it also tends to introduce overheads for enforcing isolation and mediating on requests between components. Ultimately, decomposition of a system into components requires taste and experience to understand how best to resolve such tradeoffs.

2.1 Caching and Pervasive DAC Enforcement

PFS uses α-Nexus shared memory to cache disk blocks, and the α-Nexus kernel enforces DAC for this data by associating an owner and ACL with each shared memory region. Since $PolicyMgr$ is responsible for maintaining owner and ACL information for all blocks on disk, $PolicyMgr$ is necessarily in every user TCB. So we designate $PolicyMgr$ as the owner of shared memory regions, and $PolicyMgr$ in turn adds appropriate principals to ACLs for each region on behalf of the data's owner, e.g. user IDs for blocks containing file data, or code and configuration hashes of certain PFS components for blocks containing filesystem metadata. Neither FAT32 nor partition tables have provisions for storing such policy information, so in the current implementation it is stored in a simple on-disk table accessible only to $PolicyMgr$.

In PFS, application processes and PFS components with sufficient privileges can access cached data directly by invoking shared memory `read`, `write`, or `mmap` system calls provided by α-Nexus. When an application opens a file, for example, the VFS layer obtains references to a set of shared memory pages, and VFS makes shared memory system calls to access data from those pages during subsequent file accesses.

The α-Nexus kernel implements demand paging for shared memory by making upcalls to a process designated by the shared memory owner. For PFS, $PolicyMgr$ arranges for the appropriate disk driver component to receive upcalls when pages need to be filled or flushed.

2.2 Disk Driver Components

The main task of disk drivers is to schedule and initiate DMA transfers between physical disks and the α-Nexus shared memory regions underlying the block cache. Since device drivers have large, complex code bases and many bugs, we endeavored to remove this code from user TCBs. This means preventing DMA transfers that could lead to unexpected user data accesses.

Management of multiple physical disks on a shared SATA bus requires tight coordination, whereas disks on separate busses can be managed mostly independently. To avoid excessive overhead for communication between PFS components, we did not isolate the driver code for disks sharing the same SATA bus. Instead, PFS defines a separate isolated component, DD_b, for each disk bus b. Consistent with Least Privilege, we grant DD_b privileges only for initiating DMA transfers only for disks on bus b. This means that a user storing data on one physical disk need not be at risk from a compromise of drivers for physical disks on other busses. The α-Nexus kernel includes a device driver reference monitor (DDRM) [7] that enables disk drivers to run as unprivileged processes with these kinds of fine-grained and device-specific privileges.

We extended the α-Nexus shared memory implementation so that the owner, $PolicyMgr$, can associate a tag with each shared memory page underlying the block cache. These tags were then used to extended the α-Nexus DDRM to enforce DAC for disk driver access to shared memory. For PFS, the tags specify a disk and bus number, a block number, and a status tag (e.g. *clean*, *filled*, *dirty*). Before allowing a DMA request from DD_b for some disk block (identified by disk and block number) the kernel checks that the memory address in the DMA request has been tagged by $PolicyMgr$ for holding this particular block and that the block is in the appropriate state (i.e. *empty* for DMA transfers to memory, or *dirty* for DMA transfers to disk.) Preventing DD_b from initiating arbitrary DMA requests in this way removes DD_b from user TCBs.

2.3 Filesystem Driver Components

Filesystem drivers manage file and directory metadata. A single component implementing all filesystem drivers would need to hold privileges to access blocks in every disk partition. PFS instead defines a separate, isolated component FSD_r to execute filesystem driver code for a single disk partition r. Initially, when partition r is first created, $PolicyMgr$ designates FSD_r as owner of all blocks within r. Later, when an application process P creates or enlarges a file,

FSD_r passes ownership to P for each block holding the file's data. To delete or truncate a file, P can wipe the blocks if desired before passing ownership back to FSD_r. Thus filesystem driver components managing different partitions have different privileges, and none of these PFS components have privileges for accessing the contents of files owned by applications or users.

It is tempting to further decompose filesystem drivers, since these components can be complex. For example, we could define separate, isolated components each managing a single directory. Here, the directory owner, rather than FSD_r, could take ownership of the blocks storing directory meta-data, and FSD_r could thus hold fewer privileges. We did not implement this scheme because potential benefits of such finer grained components are offset by the disadvantage of creating dependencies on the internal details of specific filesystem formats.

2.4 VFS Components

The VFS layer manages state on behalf of applications, and it translates application-level access requests into lower-level shared memory block cache access requests. For each application process P, we define a separate VFS_P component to execute VFS code on behalf of P. Thus user TCBs exclude VFS code executing on behalf of other users. The cost of running per-application VFS instances is in the overhead for managing shared VFS state. A single VFS component could manage such shared state more efficiently without inter-process communication, but a compromise of a shared VFS component could violate security goals for all users. We judged this risk unacceptable and we find the overheads for shared VFS state are small in the common case.

In the PFS prototype, the code base for VFS_P is small and simple, and it requires no more privileges than P requires. And because P likely trusts this code anyway, we execute VFS_P and P together as a single process. Actually, VFS_P was integrated into the C library used by most α-Nexus programs, making it transparent to applications. Essentially, we are trading mutual suspicion for performance, and we are leveraging fate-sharing since, even where P and VFS_P isolated, a failure of either would likely cause a failure of the other.

2.5 Credentials-based Authorization

PFS relies on access control guards provided by α-Nexus. These guards implement *credentials-based authorization* [8], a framework in which policies and credentials are specified by formulas in a formal logic. The logic supported by α-Nexus allows PFS to include a wide variety of principals on ACLs, enabling more expressive DAC policies than supported by traditional filesystems. For example, access to a file in PFS can be restricted to programs signed by a particular cryptographic key.

Credentials-based authorization is well-suited to instantiating Least Privilege internally within in PFS because it allows us to specify fine-grained principals. So an ACL in PFS can grant privileges to, say, FSD_r but not $FSD_{r'}$, since these two components are configured differently at runtime, even if both components were loaded by the same user or execute the same code. This is in contrast to approaches in which the only principals that can serve as owners or appear on ACLs are opaque, static IDs associated with users or program binaries.

PFS also relies on credentials-based authorization to restrict the privileges of filesystem administrators. In conventional filesystems, a filesystem administrator typically owns the meta-data that specifies the owner of every file and directory, and so can change the owner and, by implication, modify ACLs to gain access to any file or directory. Some conventional filesystem implementations even grant filesystem administrators, and perhaps even all programs executing on their behalf, direct and unrestricted access to all data stored on disks regardless of DAC policies. In PFS, such privileges are held by the various components comprising PFS, such as FSD_r and $PolicyMgr$, each specified by code and configuration hashes taken at runtime. In fact, PFS components hold privileges beyond those held by the administrators responsible for managing PFS. This means PFS administrators have considerably less authority: they can configure the filesystem (e.g. create, delete, mount and unmount partitions, or tune caching policies), but they do not own and cannot access user files or directories, nor can they access or modify owners, ACLs, or most other filesystem meta-data. This level of protection follows naturally from implementing complete mediation and by treating filesystem administrators no differently from other principals, given α-Nexus credentials-based authorization and the pervasive enforcement of DAC by PFS.

3 Evaluation

We performed an evaluation to measure some of the costs and benefits of the design principles we instantiated in PFS. The results—detailed below—confirm that the structure of PFS and the way it instantiates security principles brings concrete security benefits. Yet for benchmarks of several common-case operations, PFS does not exhibit markedly worse performance compared to more traditional designs.

3.1 Trusted Computing Bases

A realistic way to evaluate security benefits might be to conduct "red team" penetration testing. However, our PFS implementation—and the α-Nexus kernel on which it runs—is a research prototype written in C,

PFS/α-Nexus decomposed, user-mode	
Kernel-mode	
Sh. mem.	1,900
DDRM	4,658
User-mode	
Driver lib.	49,839
SATA driver	28,216
FAT32 driver	7,230
VFS layer	1,915
PolicyMgr	2,267
Misc.	4,562
Total Contributions	
All code	100,587
TCB for DAC	8,825
TCB for isolation	6,558

(a)

UFS/α-Nexus monolithic, user-mode	
Kernel-mode	
Sh. mem.	1,713
DDRM	4,594
User-mode	
Driver lib.	49,839
SATA driver	28,029
FAT32 driver	6,442
VFS layer	771
PolicyMgr	1,689
Misc	4,081
Total Contributions	
All code	97,159
TCB for DAC	97,159
TCB for isolation	6,311

(b)

KFS/α-Nexus monolithic, kernel-mode	
Kernel-mode	
Sh. mem.	1,673
SATA driver	28,029
FAT32 driver	6,442
VFS layer	771
PolicyMgr	1,689
Misc.	4,081
User-mode	
n/a	0
Total Contributions	
All code	42,685
TCB for DAC	42,685
TCB for isolation	42,685

(c)

Ext2:Fuse/Linux monolithic, user-mode	
Kernel-mode	
Block layer	11,051
Fuse layer	4,908
VFS layer	4,796
Partitions	3,658
ACLs	504
Common	31,348
User-mode	
Fuse layer	3,341
Ext2 driver	28,728
Total Contributions	
All code	88,334
TCB for DAC	88,334
TCB for isolation	56,265

(d)

FAT32/Linux monolithic, kernel-mode	
Kernel-mode	
Block layer	11,051
FAT32 driver	5,304
VFS layer	4,796
Partitions	3,658
ACLs	504
Common	31,348
User-mode	
n/a	0
Total Contributions	
All code	56,661
TCB for DAC	56,661
TCB for isolation	56,661

(e)

Fig. 2. Lines of code for PFS (a) and two alternative designs, UFS (b) and KFS (c), including related α-Nexus code, for Ext2 with Fuse on Linux (d), and for FAT32 on Linux (e). Totals indicate lines of code contributed to TCBs for two security goals. KFS counts are estimates. Linux counts exclude disk driver code.

so it would no doubt succumb to even a modest red-team attack. Moreover, such an exercise would mostly shed light on insecure coding practices rather than on the consequences of instantiating security principles in the design of a filesystem. So we instead examine TCB sizes, believing this to be a more useful way to gauge the utility of the PFS approach to implementing a filesystem stack.

Figure 2 details the overall size of the code base for PFS and four alternative filesystems:

- UFS, a monolithic user-mode implementation that places all filesystem code in a single component executing as a process above the α-Nexus kernel.
- KFS, a monolithic kernel-mode design that places all filesystem code within the α-Nexus kernel. Because α-Nexus does not support kernel-mode drivers, we did not implement this design but instead made estimates based on the size of various components within PFS and UFS.
- Ext2:Fuse, a monolithic user-mode implementation based on Fuse [9] and running on Linux. We used Ext2 here because a suitable FAT32 driver was not available for Fuse.
- FAT32, a monolithic kernel-mode implementation of FAT32 for Linux.

In all cases, we include only code relating to the filesystem and exclude the bulk of the kernel and operating system code base. For Linux, we also excluded disk driver code since it always executes within the Linux kernel, even for user-mode Fuse filesystems.[1]

1. The selection criteria we used is admittedly subjective, but was intended to clarify the impact of design decisions and minimize artifacts. For example, if we had counted disk driver code for Linux, as we did for α-Nexus, all Linux code bases and TCBs would appear larger by a constant amount and, consequently, it would exaggerate the benefits of PFS.

PFS has the largest code base of the designs we examined. This can be attributed mostly to a driver library, approximately 85% of which comprises a compatibility layer we implemented for running legacy kernel-mode Linux drivers in user-mode on α-Nexus. To a lesser extent, the larger code base of PFS is due to added code for inter-process communication (IPC) and authorization guards that mediate on IPC calls between PFS components. However, in contrast to the other designs, most PFS code need not be trusted because it cannot violate user data security goals even if it is compromised. To quantify this benefit, we first consider the contribution to the TCB for enforcement of pervasive DAC (or conventional DAC in the case of Linux). Since we adopted this as the primary means for protecting user data, the size of this TCB can be taken as a predictor for overall filesystem trustworthiness. Figure 2 shows that only 8,825 lines of PFS code—about 9% of the PFS code base—need be trusted by a user to enforce pervasive DAC, far less code than any other design.

Not all applications will necessarily rely on PFS, so we now consider how each design impacts the TCB relative to enforcing kernel and process isolation. The TCB for isolation is a subset of the TCB for DAC since isolation is a prerequisite for DAC enforcement in all of the designs we examined. Reducing the size of this TCB benefits all software running on the machine, because a compromise that violates kernel or process isolation can likely violate nearly any user security goal. Figure 2 shows that only 6,558 lines of PFS code need be trusted to enforce system isolation boundaries. This small TCB results from running most PFS code in user-mode. UFS has a similarly small TCB for isolation because it too places most code outside the kernel. In fact, UFS achieves an even smaller TCB here because it requires fewer and simpler exten-

sions to α-Nexus shared memory and DDRM than PFS, and because UFS authorization guards operate at a coarser granularity than in PFS. We expected a similar pattern when comparing Linux user-mode and kernel-mode designs. However, while Linux Fuse does move filesystem code out of the kernel, it extends rather than replaces most of the remaining kernel-mode filesystem stack. The net effect is that Linux user-mode and kernel-mode implementations have similarly sized TCBs for isolation.

Kernel-mode designs, represented in Figure 2 by KFS for α-Nexus and FAT32 for Linux, have the smallest code bases. This is not surprising: these designs implement only enough isolation to protect the filesystem and kernel from compromised processes but provide no support for protecting user data from a compromised filesystem. But this also means the entirety of kernel-mode filesystem implementations becomes part of all TCBs, and even applications that do not store sensitive data in the filesystem can be affected by any filesystem compromise.

Our use of FAT32 for PFS could be a concern because FAT32 is substantially simpler than more modern filesystem formats. The kernel-mode Linux Ext2 filesystem code base is about 15% larger than the kernel-mode FAT32 code base, for example, and that size roughly doubles for Ext3 and doubles again for Ext4. We believe this does not undermine our conclusions about TCB sizes. On the contrary, a larger filesystem driver code base would result in larger TCBs for α-Nexus UFS, Linux Fuse, and both kernel-mode designs, but TCBs for PFS, which are largely independent of the choice of filesystem format, would likely remain small. So using FAT32 means the results in Figure 2 likely understate the case for design driven by the security principles we adopted in PFS.

3.2 Performance

Run-time performance is an important metric, since it is often used a reason to make, rather than check, assumptions. We expect PFS to have worse performance because the added isolation boundaries and authorization guards introduce overheads. For example, PFS file `open` and `close` operations involve multiple IPC calls between PFS components, whereas each reduces to a single system call for kernel-mode designs.

To quantify performance costs associated with decomposing the filesystem stack and instantiating Least Privilege, we measured the performance of PFS on α-Nexus and compared it to the kernel-mode implementation of FAT32 on Linux. Because our PFS prototype is incomplete, we include here only results for read-intensive workloads. We implemented for four micro-benchmarks: latency for opening and closing a 1 MB file; latency for enumerating 5,461 files and directories in a 5-level tree; latency for reading a single byte from an already-open file, and throughput while reading 16 MB using a streams-oriented interface and large files with mostly sequential on-disk layout. Benchmarks were executed in an uncached configuration, in which filesystem and disk controller caches are emptied before each trial, and in a cached configuration, in which all caches are warmed before each trial and no physical disk I/O is performed.

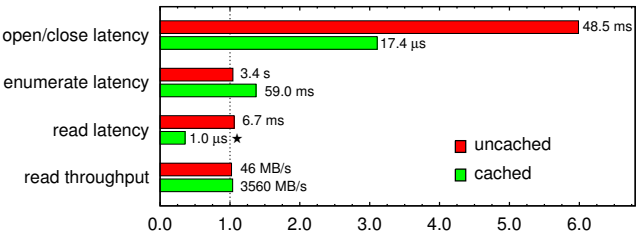

Fig. 3. Performance of PFS on four micro-benchmarks in cached and uncached configurations. Bars show normalized performance relative to a kernel-mode FAT32 implementation on Linux, numbers give median PFS result across 100 trials for each benchmark. Measured variability was low, with 80% of trials falling within ± 3% of the median, except where indicated by $\star$.

Figure 3 summarizes benchmark results. Overall, PFS read latency and throughput are comparable to that achieved on Linux, within 6% in the worst case. For uncached benchmarks, this is to be expected because performance is dictated almost entirely by the cost of accessing file data on physical disks, and any overhead due to IPC calls or authorization guards is small by comparison. The same is true for the uncached enumeration benchmark.

Cached read performance is not constrained by disk performance, so even small overheads can add up quickly. Linux processes issue one system call per `read` operation. But the same is true for PFS: once a file is open, an application process invokes the VFS layer of PFS using a local function call, and VFS can access the block cache using a single system call to the α-Nexus shared-memory subsystem. Indeed, Figure 3 shows that PFS and Linux achieve nearly identical cached read throughput performance, around 3,560 MB/s. For cached read latency, Figure 3 shows that PFS achieves a rather surprising $3x$ reduction in latency—to 1.0 µs for PFS down from 2.7 µs for Linux. We attribute this to the simplicity of the PFS user-mode VFS layer, and the α-Nexus shared memory system call it uses, when compared to the relative complexity of the Linux `read` system call and its kernel-mode VFS layer. Actually, for this benchmark, Linux showed very high variability, with 10% of trials measuring 1.4 µs or less, a value much closer to the median cached read latency of PFS.

Restructuring the filesystem stack for PFS introduces at least two potential sources of run-time over-

head: enforcing finer-grained authorization policies, and adding isolation boundaries between fine-grained components. Our benchmarks revealed cases where PFS performance is markedly affected by these overheads. For the cached open/close benchmark, median latency is 17.4 µs for PFS, approximately $3x$ worse than for Linux. Similarly, median latency for the cached enumeration benchmark for PFS is 59 ms, about 40% higher than for Linux. On both benchmarks, most of the difference is accounted for by the overhead of IPC calls between PFS components versus the performance of system calls on Linux. This difference represents the cost of isolation for fine-grained components.

The cost of enforcing fine-grained authorization policies can be significant, as evidenced by results for the uncached open/close benchmark, where latency for PFS is $6x$ higher than for Linux. In this case, enforcing pervasive DAC in PFS means that authorization policy meta-data must be checked more often and, in this benchmark, fetched more often from the physical disk. This is because PFS checks an ACL not just when an application requests to open a file, but when PFS components internally access filesystem meta-data in the course of servicing that request.

4 Related Work

Prior work considers how to design and implement secure filesystems using a variety of approaches. It is instructive to classify these approaches according to the degree of trust required between various principals that comprise and use the filesystem, e.g. users, programs, local disks, or remote storage services.

- Untrusted local disks: Many filesystems (e.g. [10], [11]) use cryptography within the kernel or in separate process to protect against theft of locally installed disks. Typically, all such filesystem code is in the TCB for DAC enforcement. Note that the system may implicitly trust local disks even if the filesystem does not explicitly do so. A disk with compromised firmware, for example, may have direct hardware access to system memory.
- Untrusted remote storage: Data can be encrypted before storing it remotely [12], [13], [14], [15], [16]. Consequently, the remote storage service holds few privileges beyond the ability to deny service, removing it from TCBs for isolation and enforcing DAC. If part of the filesystem stack executes as a process or library, or if local kernel-mode disk drivers are eliminated, then TCBs are further reduced.
- Untrusted programs: Filesystem writes can be selectively suppressed [17] or audited [18] to addresses the threat posed by programs that users do not fully trust.

CFS [10], pStore [12], TrustedDB [19], and VPFS [20] use cryptographic approaches to protect the confidentiality and integrity of data stored on behalf of one principal (e.g. an application or a user) against threats from untrusted principals executing on the same platform. VPFS additionally addresses cases where the underlying storage service is untrusted. Like PFS, VPFS involves restructuring the filesystem stack into multiple components and the design is justified explicitly on Last Privilege grounds and by considering TCBs. In VPFS, each application trusts only a shared micro-kernel and an isolated, per-application instance of the VPFS layer, which performs encryption and hash-based integrity checks on the application's behalf. Additional support code and a storage service execute in a shared, but largely untrusted, domain.

Cryptographic approaches are largely complementary to the approach we have taken in PFS. However, using encryption as an isolation mechanism requires preventing keys used for one principal's data from being revealed to other principals. This tends to limit sharing unless principals fully trust each other. Sharing files is also difficult if the filesystem itself trusts the applications that use it. pStore [12], for example, can't easily share files between users, because a substantial part of pStore executes as a library within each application's address space. A similar problem arises in TrustedDB [19], which executes application and filesystem code together without any isolation between them.

Filesystems provide a rich source of material for exploring systems that support fine-grained privilege separation. In KeyKOS [21], entire Unix-like kernels are employed to achieve isolation and reduce overlap among application TCBs. That work also explored various ways to decompose the associated KeyNIX filesystem into a collection of isolated domains. HiStar [22] similarly excludes most filesystem code from the TCB for isolation, but it introduces a custom filesystem format and uses tagged information flow as the basis of its access control mechanism.

PFS explicitly tracks policy information for disk blocks using a trusted component. The XN storage system for Exokernel [23] relied instead on untrusted but deterministic functions over disk metadata, enabling some TCBs. That allows Exokernel to remove code from some TCBs and avoid some of the costs we observed for enforcing pervasive DAC in PFS. PFS reliance on α-Nexus shared memory regions is similar to XN reliance on a trusted Exokernel cache registry and to KeyNIX use of KeyKOS segments.

PFS uses credentials-based authorization and a formal logic to enforce authorization policies. The PCFS [24] filesystem makes similar use of credentials, but authorization guards in PFS offload to applications the task of proving that authorization policies are satisfied for each request. PFS authorization guards rely on primitives implemented by the α-Nexus kernel to achieve some of the same benefits as PCFS, though α-Nexus currently lacks a general proof search procedure.

Taos [25] was pioneering in its use of an expressive logic to support authorization with fine-grained principals, and the Echo filesystem it hosted built on that support to great effect. It is not surprising then that α-Nexus primitives on which PFS relies resemble those provided by Taos: authenticated IPC channels, support for delegation, and a shared cache in which applications can store validated credentials.

Outside of filesystems, the principles guiding the design of PFS are well known, if not always followed, yet they can still be controversial. As just one example, Bernstein [26] uses Qmail to highlight the benefits of minimizing TCBs. Actually, PFS has an internal structure similar to Qmail: functionality is decomposed across numerous fine-grained components, components are mutually suspicious of each other and hold as few privileges as possible, and guards are located on the communications channels between components. But Bernstein also argues that Least Privilege is "fundamentally wrong" since it is often taken as a prescription to (i) identify existing components of the system, (ii) enumerate all privileges those components hold, then (iii) successively remove privileges so long as the system continues to function. Unsurprisingly, trustworthiness is not the outcome.

We take Least Privilege as a mandate to guide the decomposition of a system so that security-critical privileges are held by few (and small) components. While we hope that our experience instantiating Least Privilege and other security principles in the design of PFS will transfer to other systems, our approach required domain-specific knowledge to decide how to best decompose the filesystem stack. In addition to manual techniques, Buyens et al. [27] discuss a variety of partially-automated program-restructuring techniques for decomposing systems into sets of less-privileged components.

At a more fundamental level, Smith and Marchesini [28] argue that the model of subjects, objects, and privileges, upon which both Mutual Suspicion and Least Privilege rest, may no longer be adequate: is a text file or a Web page an inactive object that is acted upon, or is it an active subject that makes requests and may hold privileges?

REFERENCES

[1] M. Schroeder, "Cooperation of mutually suspicious subsystems in a computer utility," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, 1972.

[2] J. P. Anderson, "Computer security technology planning study," ESD/AFSC, Hanscom AFB, Bedford, MA, Tech. Rep. ESD-TR-73-51, Vol. 2, Oct. 1972, NTIS AD758206.

[3] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proc. IEEE*, vol. 63, no. 9, pp. 1278–1308, Sep. 1975.

[4] G. H. Nibaldi, "Specification of a trusted computing base (TCB)," MITRE Corp., Bedford, MA, Tech. Rep. M79-228, Nov. 1979, NTIS ADA108831.

[5] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider, "Logical attestation: An authorization architecture for trustworthy computing," in *Proc. ACM Symp. on Operating Sys. Principles*, Oct. 2011.

[6] G. C. Hunt and J. R. Larus, "Singularity: Rethinking the software stack," *ACM Operating Sys. Review*, vol. 41, no. 2, pp. 37–49, Apr. 2007.

[7] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider, "Device driver safety through a reference validation mechanism," in *Proc. Symp. on Operating Sys. Design & Implementation*, Dec. 2008.

[8] F. B. Schneider, K. Walsh, and E. G. Sirer, "Nexus authorization logic (NAL): Design rationale and applications," *ACM Trans. on Inform. and Sys. Security*, vol. 14, no. 1, pp. 8:1–8:28, May 2011.

[9] M. Szeredi, "FUSE: Filesystem in user-space," http://fuse.sourceforge.net/, 2012.

[10] M. Blaze, "A cryptographic file system for Unix," in *Proc. ACM Conf. on Comput. and Commun. Security*, Nov. 1993.

[11] Microsoft TechNet, "BitLocker drive encryption overview," 2010.

[12] C. Batten, K. Barr, A. Saraf, and S. Trepetin, "pStore: A secure peer-to-peer backup system," Massachusetts Institute of Technology, Cambridge, MA, Tech. Rep. LCS 632, 2001.

[13] C. A. Stein, J. H. Howard, and M. I. Seltzer, "Unifying file system protection," in *Proc. USENIX Annu. Tech. Conf.*, Jun. 2001.

[14] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: Securing remote untrusted storage," in *Proc. Annual Network and Distributed Sys. Security Symp.*, Feb. 2003.

[15] J. Li, M. N. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (SUNDR)," in *Proc. Symp. on Operating Sys. Design & Implementation*, Dec. 2004.

[16] J. Yao, S. Chen, S. Nepal, D. Levy, and J. Zic, "TrustStore: Making Amazon S3 trustworthy with services composition," in *Proc. IEEE/ACM Int. Conf. on Cluster, Cloud, and Grid Computing*, May 2010.

[17] Z. Liang, V. N. Venkatakrishnan, and R. Sekar, "Isolated program execution: An application transparent approach for executing untrusted programs," in *Proc. Annual Comput. Security Applications Conf.*, Dec. 2003.

[18] S. Jain, F. Shafique, V. Djeric, and A. Goel, "Application-level isolation and recovery with Solitude," in *Proc. ACM EuroSys European Conf. on Comput. Sys.*, Apr. 2008.

[19] S. Bajaj and R. Sion, "TrustedDB: A trusted hardware based database with privacy and data confidentiality," in *Proc. ACM Int. Conf. on Management of Data*, Jun. 2011.

[20] C. Weinhold and H. Härtig, "jVPFS: Adding robustness to a secure stacked file system with untrusted local storage components," in *Proc. USENIX Annu. Tech. Conf.*, Jun. 2011.

[21] A. C. Bomberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro, "The KeyKOS nanokernel architecture," in *Proc. USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, Apr. 1992.

[22] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, , and D. Mazières, "Making information flow explicit in HiStar," in *Proc. Symp. on Operating Sys. Design & Implementation*, Nov. 2006.

[23] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie, "Application performance and flexibility on exokernel systems," in *Proc. ACM Symp. on Operating Sys. Principles*, Oct. 1997.

[24] D. Garg and F. Pfenning, "A proof-carrying file system," in *Proc. IEEE Symp. on Security and Privacy*, May 2010.

[25] E. Wobber, M. Abadi, M. Burrows, and B. Lampson, "Authentication in the TAOS operating system," *ACM Trans. on Comput. Sys.*, vol. 12, no. 1, pp. 3–32, Feb. 1994.

[26] D. J. Bernstein, "Some thoughts on security after ten years of qmail 1.0," in *Proc. ACM Workshop on Comput. Security Architecture*, Nov. 2007.

[27] K. Buyens, B. de Win, and W. Joosen, "Identifying and resolving least privilege violations in software architectures," in *Proc. Int. Conf. on Availability, Reliability and Security*, Mar. 2009.

[28] S. Smith and J. Marchesini, *The Craft of System Security*. Boston, MA: Addison Wesley, 2007.