Costs of Security in the PFS File System*

Kevin Walsh
Department of Mathematics and Computer Science
College of the Holy Cross
kwalsh@holycross.edu

Fred B. Schneider
Department of Computer Science
Cornell University
fbs@cs.cornell.edu

July 24, 2012

Abstract

Various principles have been proposed for the design of trustworthy systems. But there is little data about their impact on system performance. A filesystem that pervasively instantiates a number of well-known security principles was implemented and the performance impact of various design choices was analyzed. The overall performance of this filesystem was also compared to a Linux filesystem that largely ignores the security principles.

1 Introduction

PFS (Principled Filesystem) pervasively instantiates several principles often proposed to guide the design and implementation of trustworthy systems: Mutual Suspicion [Schroeder 1972], Complete Mediation [Anderson 1972], Least Privilege [Saltzer and Schroeder 1975], and Minimization of Trusted Computing Bases [Nibaldi 1979]. An analysis of PFS performance thus gives insight into the impact these principles have in practice. That analysis is the subject of this paper. We observe in PFS an increase in the amount of filesystem code overall, though the trusted computing base for the filesystem—code whose compromise could result in a violation of filesystem security goals—was greatly reduced in size.

*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.

And contrary to common wisdom, we found that increased run-time overhead that could be attributed to security enforcement was not always significant.

Trustworthiness is the primary design goal for PFS, but we accept the reality that PFS might have vulnerabilities. So the design of PFS attempts to blunt the impact that a successful attack might have—on PFS, on the underlying operating system, or on applications and services unrelated to the filesystem. For comparison, conventional filesystem implementations locate the bulk of their code within the operating system kernel, where it executes in supervisor mode. The subversion of such a system would be largely unconstrained, so an attacker who successfully attacks it gains complete control over all other software running on the machine. To mitigate against such risks, PFS executes outside of the operating system kernel. Moreover, both PFS client code and PFS code itself are governed by access control policies. So damage from compromised PFS code is easier to contain.

Traditionally, filesystems enforce a form of discretionary access control (DAC) [Department of Defense 1985], where an *authorization relation* defines what accesses each *principal* is permitted to perform on each *object*. The objects include files and directories, and the principals include users and groups of users. The usual implementation of DAC employs an access control list (ACL) for each object, where the *owner* of an object is the only principal authorized to modify that ACL. The ACL enumerates the privileges each principal holds for that object.

PFS enforces DAC. But unlike conventional filesystems, PFS has distinct objects for file, directory and inode allocation lists, for disk configuration data like boot sectors and partition tables, and for (nominally) unused portions of disks. In short, PFS enforces a pervasive form of DAC in which every byte stored on a disk is part of some object and, therefore, has an owner and an ACL.

PFS Pervasive Discretionary Access Control (PDAC) Policy:

- Every sequence of bytes stored by PFS on disk has an owner.
- Every sequence of bytes stored by PFS on disk has an ACL, specified by the owner or by some principal acting on the owner's behalf, where the ACL lists principals, assigning each either `read` or `read`/`write` privileges.[1]
- A request to read or write disk data—whether on disk or in memory—made on behalf of some principal is allowed to proceed only if that principal and corresponding privilege appear on the ACL for that data.

We include among the principals the individual components that constitute PFS, so requests by the filesystem itself to access filesystem data are governed by PDAC.

As a practical matter, PFS exports a typical filesystem interface to its clients. Its internals, though, differ from that of conventional filesystems in part because PFS was designed to run on α-Nexus,[2] which exports a somewhat unconventional operating system interface.

[1] The operating system for which PFS was designed implements a separate mechanism for loading and executing code, independent of PFS. For this reason, PFS omits the traditional `execute` privilege for files.

[2] α-Nexus is a cousin of the Nexus [Sirer et al. 2011] operating system.

Access control in α-Nexus is implemented using *credentials-based authorization* [Schneider et al. 2011]. PFS both uses this authorization framework internally and exposes the underlying authorization interfaces to clients. With credentials-based authorization, policies and credentials are specified by formulas in a logic. The NAL logic used by α-Nexus supports a rich variety of principles, any of which can now serve as owners of objects or appear on ACLs. PFS consequently can implement more expressive DAC policies than those found in traditional filesystems (which typically require a file's owner and ACL entries to name users or enumerated groups of users). A group comprising a set of processes, for example, can be characterized using a predicate over program manifests—the constituents are processes that execute programs whose manifests satisfy the given predicate. This expressiveness is particularly useful for authorization of access to meta-data stored on disk, because it allows a group of PFS components, each executing as a process, to serve as an owner or to appear on an ACL.

We assume a threat model that includes (i) processes executing on the same machine as PFS and (ii) users who might ("insiders", including administrators) or might not ("outsiders") have legitimate access to that machine. PFS, however, is not designed to defend against adversaries with physical access to the system's disks, memory, processors, or other hardware.[3] PFS also trusts certain services provided by α-Nexus and trusts that α-Nexus will not itself make unauthorized requests to access filesystem data. Assurance for an operating system like α-Nexus is an independent matter.

In conventional filesystems, a *filesystem administrator* owns the meta-data that specifies the owner of every file and directory. So the filesystem administrator can change the owner and, by implication, can modify ACLs to gain access to files and directories. Some conventional filesystem implementations even grant filesystem administrators unrestricted access to all data stored on disks regardless of DAC policies. In PFS, filesystem administrators have considerably less authority. PFS filesystem administrators are authorized to configure the filesystem, but they do not own user files or directories, nor do they own most of the filesystem meta-data stored on disks. Consequently, PFS filesystem administrators are not permitted to read or write user files or meta-data, nor are they able to change owners or ACLs for user files. This protection follows naturally from treating filesystem administrators no differently from other principals, given the pervasive enforcement of DAC by PFS. Filesystem administrators in PFS can reformat disks, however, so a filesystem administrator can compromise the availability of PFS user data. Such power seems necessary for a filesystem administrator to manage the system.

The rest of this paper is organized as follows. In Section 2, we provide an overview of principles, commonly advocated for building secure systems, that we instantiate in PFS. Features of α-Nexus and NAL on which PFS depends are described in Section 3. Section 4 details the design and implementation of PFS, with specific attention to how the security

[3]Many prior filesystems (e.g., [Blaze 2003; Halcrow 2005; Microsoft TechNet 2010; Gough 2012]) do focus on defending against physical attacks, particularly off-line attacks against stolen disks. Such filesystems typically encrypt data stored on disks or make use of cryptographic hashes to monitor data integrity; PFS does not, but could.

principles of Section 2 are implemented, including trade-offs that entailed. Section 5 evaluates PFS to quantify costs and benefits of these principles. We close with a discussion of related work.

2 Security Principles

Like prior work, we view a system as a set of *components*, each with state and behavior that can be analyzed independently. Only certain prespecified components are assumed to be trustworthy relative to some set of security goals. A component whose behavior deviates from its specification is considered *compromised*; we make no assumptions about possible deviations. The smallest unit of compromise is a single component, and we do not admit the notion of a partially compromised component. Yet the behavior of even a compromised component could be constrained to violating only certain security goals. For example, a compromised component might be able to violate PDAC for some bytes on disk, but not for others.

Components cooperate by interacting. They might share state, transfer control to each other, or synchronize their behaviors. Without loss of generality, we assume that all interaction between components occurs by sending and receiving requests, responses, or other messages over prespecified channels.[4] If, by sending messages, one component can cause another to perform arbitrary actions, then the recipient is placing *full trust* in the sender. Conversely, a recipient that performs only certain actions at the request of a sender can be considered *suspicious* of the sender. Full trust may lead to violations of security goals, because it enables compromise to spread: a component that places full trust in some compromised component C may itself become compromised by virtue of receiving and acting on messages from C. Thus, suspicion is best encouraged and full trust avoided, leading to a classic security principle:

> **Principle: *Mutual Suspicion*.** Prefer designs that reduce the likelihood that any compromised component can cause the compromise of another [Schroeder 1972].

This principle can be instantiated by employing a variety of authorization mechanisms, alone or in combination. Examples include the following.

- *Restrict the language of requests and responses.* This justifies adopting an application programming interface (API) that specifies a set of interfaces and a protocol or message format understood by those interfaces. When a recipient implements such an API, senders are able to instigate only those actions defined by the API; other actions at the recipient are prevented.

- *Check incoming requests and ignore those that violate some policy.* To implement this, a *guard* (or *reference monitor* [Anderson 1972]) can be employed. The guard

[4] Familiar shared memory and method invocation semantics can be formulated in terms of sending and receiving messages over channels, as can all other types of interaction.

intercepts requests, performs checks, then forwards to the intended recipient only those requests found to comply with the policy.

- *Sanitize incoming messages before acting on them.* Web servers, for example, commonly protect against SQL injection and XSS attacks [Su and Wassermann 2006] by implementing a transformation that, for every request the Web server receives, replaces suspect character sequences with harmless ones before any further processing.

Implicit in the above examples is an assumption that all requests to recipients are intercepted. A familiar security principle summarizes this obligation:

> **Principle: *Complete Mediation*.** Authorize, using an appropriate enforcement mechanism, every request for a component to perform some action [Anderson 1972].

What obligations are entailed to enforce Complete Mediation depend on the mechanisms used to authorize requests. For guards or sanitization, Complete Mediation requires that all requests to a component be checked by an appropriate guard or be appropriately sanitized. And when Mutual Suspicion is instantiated by selecting a restricted set of APIs, Complete Mediation requires that requests must be conveyed through the appropriate API and not through some side-channel.

Complete Mediation presumes that components interact only over known, well-defined channels. Component isolation is implicit in that. An operating system typically implements isolation between processes by using a virtual memory architecture that prevents one process from accessing state associated with another process. And within a single process, software techniques (e.g., [Wahbe et al. 1993; Bershad et al. 1995]) can provide isolation between objects or code modules. In all cases, boundaries created by isolation mechanisms define the components.

If a sender can cause a recipient to perform some action, then we say the sender holds a *privilege* for that action at that recipient. When a user name is listed on the ACL for a file, for example, components executing on behalf of that user hold privileges for the corresponding file action. Privileges are sometimes a function of the system state or history in a way that is less explicit. For example, if a filesystem enforces per-process disk quotas, then whether a process has a privilege to create files depends on the process's current or past disk usage. Filesystem ACLs and disk quotas are typically enforced dynamically by guards. Privileges can also be implemented statically, for instance, by restricting at compile time the set of APIs against which a program can link.

Assurance that a system's security goals will not be violated requires analyzing all components that hold privileges for instigating any action that might violate those goals. This suggests a conservative approach to granting privileges.

> **Principle: *Least Privilege*.** Grant each component of the system the fewest privileges necessary to complete its task [Saltzer and Schroeder 1975].

Least Privilege facilitates implementing Mutual Suspicion. One way a component can cause the compromise of another is by abusing privileges that instigate actions at that other component. Eliminating (unnecessary) privileges thus helps reduce the likelihood that a compromised component will have sufficient privileges to compromise other components.

Of course, instantiating Mutual Suspicion, Complete Mediation, and Least Privilege in a system does not by itself eliminate the possibility that a system's security goals might be violated. These security principles merely help limit the impact of compromised components. Some components, by necessity, perform tasks that could violate a security goal. This set of components is called the *trusted computing base* (TCB) [Nibaldi 1979] for that security goal.[5]

Components in the TCB for each security goal must all be trusted not to violate that security goal. One way to reduce the risk that such trust is misplaced is prescribed by:

> **Principle:** ***Minimization of Trusted Computing Bases*.** Make each TCB as small as possible, consistent with the tasks it has to perform [Nibaldi 1979].

A smaller TCB should be easier to analyze; a larger TCB is more likely to have bugs, hence is more easily attacked. In an ideal system design, TCBs would be chosen to minimize the probability that security goals are violated. Minimization of Trusted Computing Bases only approximates what is sought, equating greater TCB size with increased probability of compromise.

3 α-Nexus Operating System

Several key features of α-Nexus were particularly useful for PFS. α-Nexus processes exhibit strong isolation from each other by default, making Mutual Suspicion between processes the norm rather than the exception. Processes in α-Nexus interact with each other and with the kernel by communicating over channels that have simple and straightforward semantics, chosen to support Complete Mediation. So it is trivial for an α-Nexus process to implement a guard that mediates on all incoming inter-process communication (IPC) messages. The credentials-based authorization architecture implemented by α-Nexus is useful for building guards that instantiate Least Privilege. And α-Nexus executes device drivers and various system services outside of the operating system kernel, facilitating smaller TCBs. In this section, we briefly describe how these features are used in PFS.[6]

[5] More broadly, a TCB may be construed to include not only software components, but also hardware and associated firmware. In this paper we consider only software, since PFS trusts the hardware and firmware on which it executes.

[6] A notable feature of α-Nexus is its reliance on a TPM [Trusted Computing Group 2011] secure coprocessor as a hardware-protected root of trust. Although α-Nexus relies on the attestation and secure storage facilities of the TPM to protect the confidentiality and integrity of certain data, our PFS prototype does not require a TPM.

3.1 Credentials-based Authorization in α-Nexus

In α-Nexus, each request from a principal to access a resource is accompanied by a set of credentials that convey information about relevant system state; a guard allows the request to proceed only if the credentials provide evidence sufficient to discharge the policy being enforced. Checking a policy is therefore a form of proof checking. Credentials, policies, and the names of principals are expressed using NAL [Schneider et al. 2011], and NAL provides the axioms and inference rules used to reason about credentials. In the implementation, α-Nexus provides a guard library and NAL proof checker to simplify credential management and policy enforcement. α-Nexus also implements mechanisms to ensure that credentials and requests can't be forged. For brevity, we outline a simplified treatment of α-Nexus authorization here.

A *principal name* identifies a set of one or more principals. NAL is sufficiently expressive to provide a unique principal name for each installation of α-Nexus, each process running on a α-Nexus kernel, and each user for a given α-Nexus installation.[7] Thus the principal name for a process can appear on an ACL, as can the principal name for a user.

A principal may issue credentials, and information conveyed by each credential is attributed to the principal that issued it. For instance, at the request of an executing process P, the α-Nexus kernel will issue a credential specifying a manifest for P, where the manifest describes the program code that P executes, configuration parameters for P, and so on. The resulting credential can then be used by guards when checking policies. A principal can also issue a credential to express full trust in, or delegate specific privileges to, another principal. For instance, a user $Alice$ may issue a delegation credential to establish that process P speaks for $Alice$ unconditionally. Subsequently, P can request access to resources or issue credentials on behalf of $Alice$; we need only arrange for the delegation credential from $Alice$ to accompany the request or credential that P issues. Alternatively, $Alice$ may issue a delegation credential to establish that P speaks for $Alice$ conditionally—say, only in regards to reading file f. In either case, even if only the principal name for $Alice$ appears explicitly on the ACL for f, both $Alice$ and P hold the privileges associated with that ACL entry.

A NAL *group*, denoted $\{\!|P : \phi(P)|\!\}$, is constructed from a set of principals, called *constituents*, and is specified intensionally by giving a characteristic predicate ϕ: the constituents of group $\{\!|P : \phi(P)|\!\}$ are those principals P that satisfy $\phi(P)$. NAL groups are principals, and each constituent P speaks for the group. Consider, for instance, a predicate $\phi_h(P)$ that is defined to hold exactly when $H(P) = h$, where h is the hash of some known program code and configuration and $H(P)$ is the value of a hash taken over the contents of the α-Nexus-issued program manifest credential for P. If $Alice$ places group $\{\!|P : \phi_h(P)|\!\}$ on the ACL for file f with `read` privileges, then $Alice$ has effectively granted to the group's constituents—processes executing program code having hash h—privileges to read f. In PFS, the guard enforcing this ACL would authorize a process P's request to read f only if

[7] NAL principal names are derived from user IDs (as opposed to login names) and process IDs; these IDs are never reused, even across reinstallation of α-Nexus.

it can prove that P is a constituent of that group. Such a proof would proceed most directly by presenting the appropriate program manifest credential for P from α-Nexus.

PFS defines several NAL groups as above. In the remainder of this paper, we use $HashGroup(h)$ to abbreviate NAL group $\{\!|P : \phi_h(P)|\!\}$, where h is some fixed hash value. Note that the name of the group—$HashGroup(h)$—is stable across reboots, because it is independent of the process names chosen by the kernel at run-time. So these groups are suitable for use on ACLs that are stored on disk. By contrast, process names are transient and so are unsuitable for inclusion directly on filesystem ACLs.

3.1.1 α-Nexus Alias Tables

Many authorization policies, both for PFS operations and for α-Nexus system calls, are similar in structure: the requesting principal must prove that it speaks for one of a set of prespecified principals, such as the filesystem administrator, the owner of some object, or a principal found on an ACL. Guards in α-Nexus are specialized for supporting these cases, and PFS leverages this specialization to amortize the cost incurred by guards for checking NAL proofs.[8]

As an example, suppose there are sufficient credentials to prove that a process P speaks for $Alice$, and P routinely requests access to files on $Alice$'s behalf. Once a trusted guard verifies the proof of delegation, then we can record that outcome for use up until the time when credentials appearing as premises of that proof are revoked. Similarly, once it has been proven and recorded that some predicate $\phi(P)$ holds for principal P, then it follows that P speaks for $\{\!|P : \phi(P)|\!\}$. P can thus make multiple requests on behalf of that group without incurring the cost of the guard checking that proof anew each time.

α-Nexus implements this optimization by maintaining, for each process P, an *alias table* containing a set of *aliases* $\{\ldots, A_i, \ldots\}$ and a corresponding set of proof schemas $\{\ldots, pf_i, \ldots\}$. Each proof schema pf_i would be a proof that P speaks for A_i if no credential that pf_i uses has been revoked. The first entry of process P's alias table always contains the alias P and the (trivial) proof that P speaks for itself. Thus, building a guard that enforces an authorization policy specified by an ACL is straightforward: process P specifies an index i into its alias table when making a request; the guard need only access P's alias table to check whether pf_i is a sound proof that P speaks for A_i and, if so, compare alias A_i against the principals and privileges listed on the ACL.

A process can add entries to its alias table using a system call, and other system calls delete entries, lookup and enumerate entries, etc. Much of the checking for pf_i can be performed when pf_i is added to the alias table. Later, when a lookup is performed, α-Nexus only needs to check whether credentials that are premises of pf_i are still valid, which is decidedly cheaper than re-checking the entire proof pf_i. If premises of pf_i are no longer valid, then the lookup operation fails and alias A_i is not used.

[8] The specialized guards implemented by α-Nexus and described here were replaced by a more generic type of guard in the current version of Nexus.

3.2 α-Nexus Shared Memory and IPC Channels

α-Nexus processes communicate and synchronize with each other by using shared memory regions and IPC channels. These are named using opaque identifiers, chosen at the time the region or channel is created.

- The α-Nexus shared memory API provides two access mechanisms. A process can invoke the `shm_read` and `shm_write` system calls to read or write data in a shared memory region. Or, a process can invoke the `shm_map` system call to create virtual memory mappings for the underlying memory pages—thereafter, the process can access that data directly, without kernel intervention.

- IPC channels in α-Nexus support synchronous and asynchronous message-passing semantics. A process invokes the `ipc_send` system call to send an *IPC request* over a channel and (optionally) await the corresponding *IPC response*. A process that creates an IPC channel can invoke the `ipc_recv` system call to receive IPC requests from some sending processes then, if necessary, invoke the `ipc_reply` system call to return an IPC response to that sender.

α-Nexus implements DAC for shared memory regions. Each shared memory region is owned by the process that created it, and that process controls access to the region by specifying an ACL that contains NAL principal names having `read` or `read`/`write` privileges. The `shm_read`, `shm_write`, and `shm_map` system calls take a parameter i, specifying an index into the requester's alias table. Before performing any action in response to any of these system calls, a kernel guard performs a lookup for entry i in the requester's alias table and, if the lookup succeeds, checks if the resulting alias and the requested access modes appear on the appropriate ACL. Should an owner request changes to an ACL, the kernel examines existing virtual memory mappings for all processes and deletes those found to be inconsistent with the new ACL. The kernel's shared memory guard is kept deliberately simple, because it is in the TCB for all security goals.

A process that creates an IPC channel is the owner of that channel, but α-Nexus does not implement DAC for IPC channels. Instead, Complete Mediation for IPC requests and IPC responses is achieved by implementing guards within processes that send and receive IPC requests and responses. Thus a channel owner runs a guard to authorize IPC requests arriving over the channel; other processes run guards to authorize IPC responses.

The α-Nexus kernel implements authentication for IPC channels so that guards have a reliable way to attribute IPC requests and IPC responses to the principals that sent them. The `ipc_send` and `ipc_reply` system calls take a parameter i, specifying an index into the sender's alias table. The kernel performs a lookup with that index. If the lookup succeeds, then the kernel annotates the IPC request or response with an encoding of alias A_i. Otherwise, the kernel discards the message. By annotating an IPC message with alias A_i, the kernel is attesting that principal A_i—or, rather, a process that speaks for A_i—sent the message. The recipient's guard examines IPC message annotations in the course of its own checking, and it is the sender's responsibility to ensure the alias that was selected satisfies

the recipient's guard. Because all processes place full trust in the kernel, the recipient's guard can accept and act on the kernel's annotations in lieu of performing is own proof checking.

3.3 α-Nexus Device I/O Privileges

Device drivers in α-Nexus run as processes above the kernel. Each physical I/O device is associated with a device driver process. The process and device interact using a set of (unique) I/O addresses associated with the device; device drivers are not authorized to access other I/O addresses. Device drivers request I/O to instigate direct memory access (DMA) transfers between system memory and devices, but they are not normally authorized to request I/O that causes DMA transfers to memory outside of the device driver process's virtual memory or to memory that is otherwise unsuitable for DMA transfers.[9]

Neither the kernel nor other processes place full trust in device driver processes. α-Nexus implements a guard, called the *device driver reference monitor* (DDRM) [Williams et al. 2008], that tracks relevant system state (and history) in order to distinguish between safe and unsafe I/O operations. Whether a particular I/O operation is permitted by the guard depends on system state including, for example, current DMA-compatible memory allocations, the history of I/O operations previously requested, and the identity and state of the device to which the I/O is addressed.

Complete Mediation requires that the DDRM be invoked for every I/O request. To implement this, a device driver process makes system calls to request that the kernel execute I/O instructions on the process's behalf; α-Nexus disables native hardware I/O instructions for all device driver and other processes. The α-Nexus kernel invokes the DDRM before performing any action in response to a device driver process's I/O-related system calls. In order that the DDRM can accurately check whether a requested I/O operation is safe, the kernel also notifies the DDRM of relevant changes to system state, such as when DMA-compatible memory is allocated or deallocated.

The DDRM is an example of Mutual Suspicion using an *external guard*—a guard that is implemented not by the intended recipient of requests (i.e., the hardware device) nor by the channel that conveys requests (i.e., the device I/O mechanism), but by some other principal. Using an external guard increases the costs associated with enforcing a policy. In this case, I/O operations incur extra latency by involving the kernel instead of being executed directly by the device driver process. Using an external guard was necessary here because the recipient, a physical device, does not implement a guard and can't be easily modified to do so.[10]

[9] Memory pages used in DMA transfers must be properly aligned, have virtual memory paging disabled (i.e., "pinning" the pages in memory), and be physically contiguous with low physical addresses.

[10] Some hardware platforms include support for enforcing policies in hardware similar to those enforced by the DDRM. These platforms partially obviate the need for the DDRM.

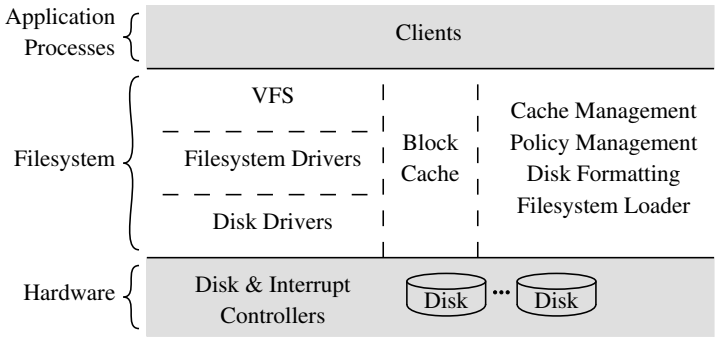

Figure 1: Structure of the PFS filesystem.

4 Design and Implementation of PFS

The basic design and some of the code for PFS derive from a Linux filesystem [Linux Kernel (version 2.6.29.2) 2009]. PFS is organized into three main layers, augmented by several code modules for concerns that cut across multiple layers. The layers and major modules are shown in Figure 1.

- A bottom layer comprises *disk drivers*. These manage DMA transfers between system memory and disks. The disk drivers are based on the Linux libATA [Garzik 2012] driver library.

- A middle layer comprises *filesystem drivers*, each implementing a standard file and directory tree view of the data stored in one partition of the disk. We implemented one filesystem driver for use in PFS; it is an adaptation of an open source FAT32 filesystem driver [Riglar 2010].[11]

- A top layer creates a virtual filesystem (VFS). VFS presents a single file and directory namespace to clients, and it hides some details of filesystem drivers. The VFS interface for clients includes a *streams-oriented* interface, in which a client invokes `read` and `write` methods to access file contents, and an *mmap-oriented* interface, in which file contents are mapped into a client's virtual address space for subsequent direct client access.

PFS runs on standard Intel x86-compatible hardware with dual Serial ATA (SATA) disk buses, each capable of controlling between one and four physical disks.

PFS data is stored on disks in 4,096-byte blocks; this size coincides with the memory page size on our hardware. The k^{th} block on the j^{th} disk of disk bus b is uniquely identified by the tuple $\langle b, j, k \rangle$, called a *block ID*. Block 0 of each disk stores a *partition table* for

[11] We chose the FAT32 format for ease of implementation. Other filesystem formats, such as EXT3 or NTFS, would also be straightforward to deploy by adapting available drivers.

that disk. Each entry in the partition table defines a partition, specifying a *type code* and a range of block numbers. The type code identifies a filesystem driver that manages the corresponding range of blocks (our prototype only supports one type code, FAT32, because it only implements that one filesystem driver). A PFS disk-formatting module writes a new partition table to block 0 once, during system installation, and the PFS filesystem-loader module reads this block during each reboot.

The owner and ACL for each block are stored on disk and thus persist across reboots. PFS does not impose constraints on what NAL principal names are used to specify owners or ACL entries, so PFS benefits from NAL's flexibility. Typical principals include users, named sets of users (i.e. traditional enumerated groups), and NAL groups whose constituents are processes. This authorization information is first written during initial system installation, and PFS flushes changes to disk whenever a block's ACL is modified or the block's owner changes.

4.1 File Access Requests

It is instructive to consider the sequence of events that occur when a client reads a file. Suppose some client sends VFS a request, specifying a *file handle* and a count of bytes to read.

1. Upon receiving the request, VFS looks up the file handle in a per-client `file_descriptor_table` to obtain that client's current offset for file accesses and a unique ID for the file. The unique ID is a pair, comprising a reference to the underlying filesystem driver for that file and an ID chosen by that filesystem driver. VFS calculates the range of bytes to be read then forwards a request, with the byte range and unique ID, to the appropriate filesystem driver.

2. A filesystem driver, upon receiving such a request, translates the file ID and range into a set of block IDs. This computation can require access to meta-data stored on the disk or cached in memory. Once the block IDs are computed, the filesystem driver checks if the requested blocks are cached. If the blocks are not, then the filesystem driver sends the block IDs in a data transfer request to the appropriate disk driver.

3. A disk driver translates each such request into a series of I/O requests. A hardware interrupt on completion of the transfer initiates a completion action at the disk driver. The disk driver then notifies the filesystem driver where the data can be found in the cache.

4. The filesystem driver likewise notifies VFS where the data can be found in the cache.

5. The VFS layer updates the current file offset in the client's `file_descriptor_table` then passes the data from the cache to the waiting client.

Filesystem drivers communicate with disk drivers through a shared `io_request_queue` data structure that contains data transfer requests to be serviced (asynchronously) by the

disk driver. Filesystem drivers insert new requests into this data structure, and they poll to check whether previously inserted requests have completed. Requests are not necessarily performed first-in, first-out, because disk drivers sometimes re-order requests, combine overlapping requests, or aggregate requests for nearby blocks.

4.2 Caching

To mask the high latency and low throughput of physical disks, PFS caches disk blocks in memory. A PFS cache management module tracks the status of cached blocks, and it monitors global memory usage and block access patterns. A cache replacement algorithm makes decisions about which cached blocks should be flushed or evicted from the block cache, as well as which blocks on disk should be preemptively loaded into the block cache. Our prototype implements most of the cache replacement strategy as part of the cache management module, since this module has comprehensive information about current and past cache usage across all filesystems; other arrangements are possible.

PFS avoids the expense of copying blocks when requests or responses move between layers of the filesystem. Thus, messages within and between layers can refer to the unique copy of data stored in the block cache; all parts of the filesystem share access to the block cache. Disk hardware also shares access to cached blocks, performing DMA transfers directly between the block cache and disks.

4.3 Heuristics for Decomposing PFS

While the basic organization of PFS, as described above, is dictated by functionality requirements, there was considerable flexibility about how to decompose PFS into components. Clearly, we should prefer small, fine-grained components, since that offers more opportunities to instantiate Mutual Suspicion, Least Privilege and Minimization of Trusted Computing Bases. Were PFS implemented as a single large component, for example, then the entire code base for PFS would be in the TCB for all security goals. If, instead, PFS is decomposed into small components, each with only a small amount of state and code, then each component conceivably could be granted only a relatively small set of privileges. Some components might even be excluded from some of the TCBs.

PFS is decomposed into fine-grained components, according to the following heuristics.

- *Domain decomposition* [Foster 1995]: Define a single component for each cohesive subset of the system state, and include in that component all code necessary for managing and manipulating this state. For example, file descriptor tables containing information about open file handles might be assigned to one component; and partition tables describing disk layouts might be assigned to another component. If some security goal concerns only a portion of the system state (e.g., file descriptors or partition tables), then this decomposition helps minimize TCB size for that goal. However, the price of this organization is that a single task that involves much of the system state now involves interactions between many components.

- *Functional decomposition* [Foster 1995]: Define a distinct component for each separate task. Such a decomposition often results in components whose boundaries coincide with natural units of code. For example, several code modules in PFS are each implemented as independent components, including: $DiskFormatter$, to execute the disk-formatting module; $FSLoader$, to execute the filesystem-loader module; $CacheMgr$, to execute the block cache management module; and $PolicyMgr$, to execute code for managing block owners and ACLs. As with domain decomposition, functional decomposition helps minimize TCB size by isolating critical functionality from unrelated parts of the system. However, functional decomposition can require replicating or sharing data when tasks being assigned to different components use the same data.

- *Privilege separation* [Kilpatrick 2003; Provos et al. 2003]: Decompose code into components according to privileges, placing code requiring similar privileges in the same component and placing code requiring different privileges in different components. This approach should result in designs that allow many opportunities to instantiate Least Privilege. But the approach presumes that privileges are defined before the system is decomposed. In PFS, this was the case only for privileges associated with device drivers that are enforced by the DDRM.

In applying the above heuristics, we must consider the impact on TCB sizes. For instance, certain PFS components are in the TCB for PDAC enforcement because they are responsible for protecting the integrity of the enforcement mechanisms for this policy. Consistent with Minimization of TCBs, we endeavored to reduce the number and size of such components. Execution of all other PFS components is governed by that enforcement mechanism. In particular, those components are prohibited from accessing blocks (cached in memory or stored on disk) that contain contents of user files—those blocks are owned by users, so they do not include PFS components on their ACLs (an instance of Least Privilege). We instantiate Least Privilege for blocks storing other information, as well: The owner of each such block is the PFS component that creates and manages the block's data, and a PFS component is included on the corresponding ACL only if that component needs to access that data to complete its task. In fact, PFS components rarely share access to block data, so most ACLs for these blocks contain only a single entry—an entry granting the owner of the block full access.

Granularity of a decomposition affects performance, so heuristics must be applied carefully. A run-time cost is associated with supporting isolation for a component, so in designs with many finer-grained components, the total costs for implementing isolation will be larger. System performance can also suffer, due to overhead for supporting interaction between a larger number of components. This overhead includes the cost of provisioning channels, the extra cost to communicate over such channels as compared to using shared memory, and the costs incurred by mechanisms for Least Privilege, Complete Mediation and Mutual Suspicion concerning messages sent over channels. PFS locates most components outside the kernel, where each component executes as a separate process. In some

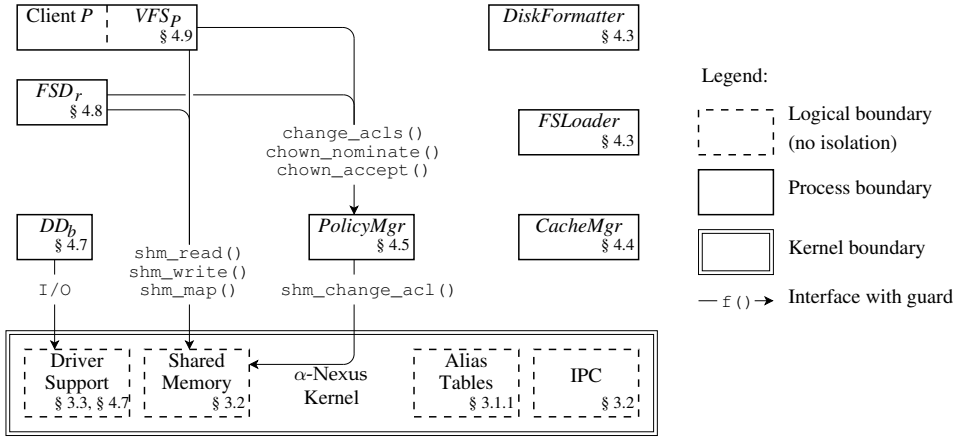

Figure 2: Final implementation of the PFS filesystem, showing major components and some of the interfaces between components. Numbers refer to the sections of this paper in which each component or interface is discussed.

cases, we judged the higher performance costs associated with supporting process isolation to outweigh its contribution to trustworthiness.

One such case is discussed in Sections 4.11.2 and 5.3. There, more than one PFS component is executed in a single process, even though this means sacrificing some isolation. In another case, discussed in Section 4.7, a small amount of PFS code is situated within the kernel, thereby enlarging the TCB for all security goals. All else equal, a filesystem decomposed into many fine-grained components ought to exhibit worse performance than one decomposed into fewer coarse-grained components.

Finally, the three heuristics described above can conflict. Consider the `io_request_queue` data structure implementing a list of pending disk transfers. This data structure is manipulated both by I/O scheduling code and by disk driver code. Domain decomposition would suggest placing both the I/O scheduling code and disk driver code into a single component, because they share state. But privilege separation would suggest having two separate components, one for the scheduler and another for the disk driver, because only the disk driver code requires privileges to perform disk I/O. In the end, decomposition of a system into components requires taste and experience to understand how best to resolve such conflicts.

Below, we describe the aforementioned PFS components in some detail, justify how the remainder of PFS was partitioned into components, and describe the privileges each PFS component holds. Figure 2 provides a guide for the discussion by illustrating the final structure we arrived at for the PFS implementation, including the major components of PFS and some of the interfaces between those components.

4.4 Cache Management Component

For each page of memory allocated for the block cache, $CacheMgr$ tracks: a block ID, a reference count, usage statistics, and a *page status flag*. The page status flag can be either `empty`, for a page that is allocated but not yet filled with data, or `filled`, for a page that has been filled with data for the appropriate block from disk. In the later case, a *block status flag* is either `dirty`, for a page containing cached block data that has been modified in memory and not yet flushed to disk, or `clean`, otherwise. Block and page status flags are used to ensure that clients and PFS components do not access a page in the block cache before it is filled with appropriate data. The flags also help ensure that changes made to cached blocks are flushed to disk before block cache memory pages are deallocated.

PFS components and clients refer to cached blocks using *cache references*. Cache references are passed as parameters when invoking the $CacheMgr$ API. That interface is an IPC channel that allows incrementing and decrementing reference counts, reading or updating usage statistics, and reading block and page status flags. Because efficient access to cached blocks is so critical to system performance, PFS stores the contents of cached blocks in α-Nexus shared memory regions. A single ACL controls access to all blocks in a single shared memory region, so blocks with different owners or different ACLs are stored in different shared memory regions. For example, a separate shared memory region is created for each file. This enables PFS to instantiate Least Privilege for access to blocks.

Cache references are pairs comprising a shared memory region ID and a page offset. A process accesses the contents of a block by first extracting the shared memory region ID from the cache reference, then invoking interfaces in the α-Nexus shared memory API: for mmap-oriented access, a process invokes the `shm_map` system call; for streams-oriented access, a process invokes `shm_read` or `shm_write` system calls.[12]

When a process requests access to data in an α-Nexus shared memory region, the requester provides the absolute offset and length of the data sought. For a client that uses streams-oriented access to files, VFS calculates these offsets then invokes `shm_read` or `shm_write` system calls on behalf of the client. For a client that uses mmap-oriented access to files, the client calculates the needed offsets and the client invokes `shm_map` directly.

Cache references rather than block contents typically appear in messages between processes; and processes access and manipulate blocks indirectly through system calls. This indirection can improve performance by avoiding copying. And the interposition of an API for accessing and manipulating blocks, in effect, exposes only a limited set of operations (hence, privileges) as compared to a design in which block data is copied between processes and accessed directly by processes. For example, $CacheMgr$ holds privileges to initiate prefetching and eviction for blocks, but $CacheMgr$ does not hold privileges to read or write cached blocks. And a disk driver can't directly read or write blocks stored in the cache or on disk, even though it holds privileges for initiating DMA transfers between the block cache and disks. Thus, our use of indirection enables PFS to better instantiate both

[12] PFS's use of α-Nexus shared memory regions is similar to a more traditional filesystem implementation's use of a kernel buffer cache or page pool.

Mutual Suspicion and Least Privilege.

4.5 Policy Management Component

The PFS *PolicyMgr* component manages *policy meta-data* comprising the owner and the ACL for each block stored in PFS.[13] For each installed disk, *PolicyMgr* maintains a `policy_table` data structure to store policy meta-data for that disk. Each entry in `policy_table` encodes a range of block IDs, a NAL principal name for the owner of those blocks, and an ACL enumerating NAL principal names and the privileges those principals hold.

PDAC dictates that access to a block—whether stored in the block cache or on disk—is allowed only if the requester appears on the appropriate ACL. Blocks stored on disk can only be accessed using disk transfers, so PFS authorizes disk transfers only in certain limited situations, e.g., to flush a cached block to disk or to load a disk block into the block cache. The mechanism for enforcing these restrictions is described in Section 4.7. For blocks stored in the block cache, the shared memory guard that the α-Nexus kernel provides is sufficient for enforcing PDAC; we need only configure each shared memory region's owner and ACL. *PolicyMgr* creates the shared memory regions used for the block cache, hence the *PolicyMgr* process is the owner for these shared memory regions and, consequently, *PolicyMgr* can configure each shared memory region's ACL. *PolicyMgr* determines the contents of that ACL by reading `policy_table` for the blocks that are expected to be stored in the shared memory region. All those blocks must have the same ACL, otherwise *PolicyMgr* will not create the shared memory region.

Each `policy_table` is stored on the corresponding disk, and therefore policy meta-data persists across reboots. Portions of a `policy_table` are cached in memory and used when *PolicyMgr* creates a shared memory region for the block cache. *PolicyMgr* uses the same mechanisms to read or write this meta-data on disk as used for all other disk accesses in PFS. So the cached `policy_table` is actually stored in the block cache. This architecture makes bootstrapping PFS a bit tricky since, as previously described, PFS prohibits reading disk blocks except to load the block cache, but *PolicyMgr* must read `policy_table` when creating the shared memory regions that hold cached blocks. We resolved this circular dependency by storing `policy_table` at a fixed, predetermined location on disk. On reboot, *PolicyMgr* knows which disk blocks store `policy_table`, so it creates a shared memory region to cache those blocks along with an ACL that grants only itself access to that shared memory region. *PolicyMgr* can then request that blocks storing `policy_table` be loaded from disk into the block cache, as needed, before creating additional shared memory regions for other blocks. Thus, in effect, ACLs for blocks storing a `policy_table` are implicit in PFS.

Requests to change block ACLs *PolicyMgr* implements an API for changing the ACL

[13] Many filesystem drivers contain code to manage policy meta-data for the files and directories that they manage. However, the FAT32 filesystem does not directly support DAC and makes no provision for storing policy meta-data within a FAT32 partition, so such code was not present in the filesystem driver we adapted.

associated with a range of block IDs. A process can invoke a `change_acls` method over an α-Nexus IPC channel, specifying the range of block IDs and how the ACLs should change, e.g., modify the privileges in an existing ACL entry, add a new ACL entry, or delete an existing ACL entry. In response to a `change_acls` request, $PolicyMgr$ updates the ACLs in the cached copy of `policy_table` in memory and flushes the changes to disk. If any of the modified ACLs concern blocks in the block cache, then $PolicyMgr$ also invokes the kernel to update the kernel-maintained ACL associated with the shared memory region for those cached blocks. Thus $PolicyMgr$ ensures all copies of an ACL are consistent.

According to PDAC, only a block's owner—or some principal that speaks for the block's owner—may change an ACL. So $PolicyMgr$ implements a guard to authorize `change_acls` requests. The guard allows a `change_acls` request from some process P to proceed for some block ID d only if there is a proof that P speaks for $owner(d)$ with regard to `change_acls` requests for that block. Here, $owner(d)$ denotes the NAL principal name for block d's owner, as found in `policy_table`. PFS allows process P to accompany the `change_acls` request with the necessary proof and the guard invokes NAL's automated proof checker to check that proof.

Since each `change_owner` request is conveyed over an α-Nexus IPC channel, $PolicyMgr$ leverages α-Nexus alias tables to amortize the cost of proof checking. Before invoking `change_acls`, P installs $owner(d)$ in its alias table along with a proof that P speaks for $owner(d)$; P then specifies an index for the resulting alias when sending IPC requests to $PolicyMgr$. The $PolicyMgr$ guard compares the NAL principal name that accompanies each such IPC request against $owner(d)$ for each block ID d in the specified range. If any of these comparisons fail to match, then the `change_acls` request is denied.

Requests to change block owners $PolicyMgr$ implements an API for changing the owner for a range of block IDs. In a traditional filesystem, the `chown` system call changes a file's owner, and typically only a filesystem administrator is allowed to invoke `chown` on a file. PFS enforces PDAC for the filesystem administrator and does not have the filesystem administrator speak for all users or processes. So $PolicyMgr$ enforces a different policy, one that does not involve the filesystem administrator: before changing the owner of a block d to some new NAL principal name A, PFS requires the current owner and new owner to both consent to the change in ownership. $PolicyMgr$ implements two methods—`chown_nominate`(d, A) and `chown_accept`(d)—that processes can invoke over an IPC channel.[14] PFS updates `policy_table` with new ownership information only if `chown_nominate`(d, A) is invoked by a process that speaks for $owner(d)$ and `chown_accept`(d) is separately invoked by a process that speaks for A. Thus, a change in ownership requires coordination between the current owner, $owner(d)$, and a proposed new owner, A. This coordination is done out of band.

PFS leverages the α-Nexus alias table for processing requests to change a block's owner. The protocol is as follows. First, a process executing on behalf of the current owner adds

[14] For simplicity of presentation, we show these methods as accepting a single block ID d. In practice, the methods accept ranges of blocks IDs.

$owner(d)$ to its alias table and sends an IPC `chown_nominate`(d, A) request to $PolicyMgr$, specifying a block ID d and a proposed new owner A. Upon receipt of this request, a guard checks that the NAL principal name accompanying the IPC request equals $owner(d)$. If so, $PolicyMgr$ records A in a temporary variable, denoted $nomination(d)$, for later use. Subsequently, a process executing on behalf of the proposed new owner adds A to its alias table and sends an IPC `chown_accept`(d) request using that alias to $PolicyMgr$. Upon receipt of this request, a guard checks that the NAL principal name that accompanies the IPC request equals $nomination(d)$. If so, both the current and proposed new owner have consented to the change, and $PolicyMgr$ records $nomination(d)$ as the block's new owner in $policy\_table$, flushing the change to disk. At most one instance of $nomination(d)$ is stored for each block d, and these variables are discarded on reboot.

Complete Mediation for policy meta-data PFS could store as many as three copies of policy meta-data: (i) encoded in `policy_table` on disk, (ii) encoded in a cached copy of `policy_table`, and (iii) in the ACLs for shared memory regions that form the block cache. Copies (i) and (ii) are accessed using the same mechanisms as used for all other blocks, so PFS's normal PDAC enforcement mechanisms suffice for implementing Complete Mediation: PFS assigns $PolicyMgr$ to be the owner of blocks storing `policy_table`, and $PolicyMgr$ specifies an ACL containing only itself for those blocks. Copy (iii) is stored within the kernel, and the kernel's shared memory guard allows only the owner of the corresponding shared memory region to change it. And because $PolicyMgr$ is the owner of the shared memory regions that form the block cache, only $PolicyMgr$ can directly change copy (iii) of the policy meta-data. $PolicyMgr$ makes changes to any of the copies only in response to requests that its own guard has authorized, i.e., after checking the appropriate policy for a request to change a block's owner or ACL, as described above.

4.6 TCB for PDAC Enforcement

$PolicyMgr$ and $CacheMgr$ are both involved in managing cached blocks. We considered incorporating all cache-related code into a single component, for better performance. But we rejected that design after considering its impact on the TCB for PDAC enforcement. Code for managing policy meta-data is in this TCB, but most cache management code need not be. For example, code that implements cache prefetching does not need privileges to modify ACLs. The current PFS decomposition, which involves two components, allows a smaller, simpler TCB. We (correctly) predicted that the performance costs of having the two separate components would be acceptable, given the infrequency of block allocation.

Because $PolicyMgr$ is in the TCB for PDAC enforcement, we included in $PolicyMgr$ the remaining PFS code necessary for enforcing PDAC. $PolicyMgr$ makes sure dirty cached blocks are flushed before they are deallocated, for example, and $PolicyMgr$ ensures that each block can be found at most once in the block cache to avoid cache aliasing issues. The result is that, aside from the α-Nexus kernel, $PolicyMgr$ (2,267 lines of C code) is the only PFS component in the TCB for PDAC enforcement.

4.7 Disk Driver Components

PFS uses a component DD_b to execute disk driver code for each disk bus b. This decomposition is an example of privilege separation, because components are being defined based on security-relevant privileges they require—in this case, privileges to request I/O for a given disk bus. We considered decomposing disk drivers into finer-grained components by using a separate component for each disk rather than for each disk bus. But this decomposition required the DDRM to distinguish I/O requests on a per-disk basis, and α-Nexus did not support that.[15]

DD_b holds privileges for initiating DMA transfers between disks on bus b and the block cache. PDAC requires that only certain transfers be allowed. A straightforward, but ultimately unsatisfactory, approach to restricting access to blocks on disk is to include a guard in each disk driver DD_b. This implements Complete Mediation, because only DD_b can request I/O operations for disk bus b. But this approach also puts DD_b in the TCB for PDAC enforcement, a bad idea given the size, complexity, and (historically) high rate of bugs in device driver code [Chou et al. 2001].

The approach we implemented in PFS was to extend the DDRM with checks to enforce PDAC. This implements Complete Mediation, because all device I/O requests are checked by the DDRM. This approach also helps to minimize the TCB for PDAC enforcement, since that TCB already includes the kernel, hence the DDRM, and we need only add to the DDRM a small amount of code. Because the added code is small, other TCBs, which necessarily include the kernel, are not affected much.

In addition to the regular DDRM checks, for I/O operations that instigate a DMA transfer between block having ID d and memory page m, the DDRM checks whether:

(i) page m is within some shared memory region r, owned by some process P;

(ii) P speaks for $PolicyMgr$;

(iii) page m stores, or is expected to store, block d;

(iv) if transferring to memory, then page m is `empty`; otherwise, if transferring from memory, then page m is `filled` and `dirty`.

We modified α-Nexus shared memory code so that shared memory allocations for the block cache satisfy hardware constraints for DMA transfers. And we modified the DDRM to update the associated page status flag to `filled` (if it was not already) and the block status flag to `clean`.

The DDRM implements (ii) by searching P's alias table for an entry encoding a NAL principal name for $PolicyMgr$ accompanied by a proof that P speaks for $PolicyMgr$. If no such alias table entry is found, then the DDRM denies the I/O request. We define $PolicyMgr$ to be the NAL group $HashGroup(h_{PolicyMgr})$, where $h_{PolicyMgr}$ is the hash of a

[15] Attributing I/O requests to specific disks introduces significant dependencies on the specific type of disk controller hardware being used.

program manifest describing the PFS code for $PolicyMgr$. So a process executing the code for $PolicyMgr$ will have $h_{PolicyMgr}$ listed in its α-Nexus-issued program manifest credential. The process is thus a constituent of the group and can prove, using that credential, that it speaks for $PolicyMgr$. Using a NAL group in this way provides stability across reboots by allowing the DDRM to ensure that the shared memory region's owner is $PolicyMgr$, even though the NAL principal name for the process P that executes $PolicyMgr$ changes on each α-Nexus reboot.

Checks (iii) and (iv), above, depend on meta-data associated with each page in the block cache. We considered two implementations.

- The meta-data could be stored in a PFS component, requiring the kernel to access some process's memory whenever the kernel needs to access the meta-data.
- When $PolicyMgr$ invokes the kernel to create a shared memory region, it specifies the block IDs for each page (page status flags are always `empty` initially), and the kernel stores and manages the meta-data thereafter.

We chose the later, because a component that manages the meta-data would become part of the TCB for PDAC, whereas the kernel is already part of this TCB. Moreover, the kernel code to manage the meta-data is actually simpler, so likely less error-prone than the code needed for the kernel to access meta-data that is stored and managed by a process. Finally, the kernel accesses this meta-data more frequently than other components, so it pays to locate the meta-data within the kernel.

4.8 Filesystem Driver Components

The filesystem driver layer manages file and directory meta-data for the filesystem stored by each disk partition. A single component implementing all filesystem drivers would have to hold privileges to access blocks in every disk partition, which is inconsistent with Least Privilege. So we used privilege separation and decomposed this layer into multiple components. In PFS, each disk partition r has a separate component FSD_r. FSD_r executes the filesystem driver code given by that partition's type code, and FSD_r is configured with the range of block IDs that define that partition.

When a disk in PFS is first formatted, the $DiskFormatter$ component becomes owner of all blocks on the disk. $DiskFormatter$ creates a partition table and, for each partition r in the partition table, invokes `chown_nominate` to propose a FSD_r as the new owner for the blocks in that partition. FSD_r then invokes `chown_accept` and becomes the owner of these blocks. For stability across reboots, the NAL principal name used for FSD_r is the NAL group $HashGroup(h_{FSD_r})$. As with $PolicyMgr$, the group definition includes a hash h_{FSD_r} computed over a program manifest, where the program manifest describes the code being executed by FSD_r. But the manifest here also includes the range of blocks IDs that define partition r, so that filesystem driver components configured to manage different partitions will have different NAL names, hence will hold different privileges, even if the components execute the same filesystem driver code.

The top layer of the filesystem, VFS, employs an IPC channel to invoke FSD_r for various file and directory operations. When a process P, on behalf of principal A (e.g., a user), requests that a file be created or enlarged, FSD_r invokes `chown_nominate` to propose that A become owner of blocks that will store contents of that file. Process P then invokes `chown_accept` to become owner of those blocks. These roles are interchanged when a file is truncated or deleted. Once the owner of the file becomes the owner of blocks storing the contents of the file, the α-Nexus shared memory guard and $PolicyMgr$ enforce PDAC for those blocks. Thus FSD_r is not in the TCB for PDAC enforcement.

Users and other principals own the contents of files, but FSD_r owns other blocks in partition r, including blocks that store directories. We could have further decomposed PFS. One approach would employ a separate component for directory management code. Or, using domain decomposition, we could have separate instances of directory management code for different directories into different components. These alternative architectures could allow users or other principals to own blocks storing directories, thereby permitting FSD_r to hold fewer privileges. The potential benefits of finer grained components, however, are offset by the disadvantage of creating dependencies on the particular filesystem format—FAT32—used in our prototype.

4.9 VFS Components

The VFS layer manages state on behalf of filesystem clients. One VFS_P component for each client P comprise the layer. Client P invokes VFS_P to perform various file and directory operations, and VFS_P implements a guard to ensure requests only from P are performed.

Client P always executes on behalf of some principal A, usually a user or a group of users. In PFS, A trusts P fully by default, and A issues a delegation credential to this effect. P can therefore add A to its alias table, using A's delegation credential to prove that P speaks for A. Client P issues a credential that further delegates privileges to VFS_P; taken together with the credential from A, this allows construction of a proof that VFS_P speaks for P, hence a further proof that VFS_P speaks for A as well. Thus VFS_P can also use A as an alias and make requests to filesystem drivers and to α-Nexus shared memory regions on behalf of A.

If client P uses an mmap-oriented interface to access files, then P must invoke the `shm_map` system call directly, rather than having VFS_P make the request on its behalf. This is necessary on α-Nexus, because the kernel creates the virtual memory mappings in whichever process invoked the system call. In this case, P uses alias A to satisfy the kernel's shared memory guard, since A is presumably on the ACL for the blocks in question.

Using a separate VFS_P component for each client P helps reduce the TCB size for any security goal a client might have. If VFS_P becomes compromised, then security goals relating to client P may be violated (recall that P has placed full trust in VFS_P by issuing a credential to that effect), but some other client P' not having placed any trust in VFS_P would be unaffected by the compromise. Were the VFS layer a single component, then a

compromised VFS layer could cause security goals to be violated for all clients that use PFS.

Even though VFS_P is responsible for specifying the identity of A to other components—by adding alias A to its alias table and selecting that alias when sending IPC requests—VFS_P need not be trusted by the recipients of these requests, because the kernel checks that P indeed speaks for A during each `ipc_send` system call. So absent a proof that VFS_P speaks for A', for some other principal A', VFS_P can't substitute a bogus identity A' when making requests, even if VFS_P is compromised. By contrast, were the entire VFS layer a single component, then VFS would need to be trusted by other PFS components to chose the right identity from among many valid alias table entries when making a request on behalf of a client.

In our PFS implementation, each client P places full trust in VFS_P. Least Privilege would favor a design in which P does not place full trust in VFS_P, but instead VFS_P is granted only those privileges needed to perform its task—only privileges to make requests to various PFS components and α-Nexus shared memory regions on behalf of P. There are several ways to implement this design using NAL, but we found it unnecessary given optimizations described in Section 4.11.2.

4.10 Data Replication in PFS

In our decomposition of PFS into components, certain data is used by multiple components. The VFS components share a `mount_table` data structure containing information about each mounted filesystem. And filesystem driver and disk driver components share `io_request_queue` data structures. A straightforward implementation would use α-Nexus shared memory to store these shared data structures. But this restricts the granularity of authorization we could enforce, since α-Nexus shared memory supports only two types of privileges, `read` and `read`/`write`, and it enforces these at a relatively coarse granularity (4KB memory pages). Least Privilege would favor a design where PFS components hold fewer and more fine-grained privileges. For example, while VFS components executing on behalf of the administrator can create or delete entries in `mount_table`, other VFS components only read the entries and increment or decrement various reference counts. Similarly, only filesystem drivers insert entries into an `io_request_queue`, and only disk drivers mark those entries as completed.

We avoid limitations associated with authorizing access to a common data structure by replicating information across different data structures. Different privileges are then associated with the different replicas. PFS employs this approach, using α-Nexus IPC channels to keep the replicas synchronized. For `mount_table` and `io_request_queue` data structures, straightforward coherence protocols suffice to accommodate the different privileges held by the components that need access. The performance cost of this approach to Least Privilege is the overhead for implementing a coherence protocol for the replicas.

4.11 PFS Implementation Optimizations

In building PFS, we explored a few techniques that improve performance without sacrificing benefits to system trustworthiness brought by instantiating security principles. The optimizations we considered generally admit slightly less aggressive instantiations of security principles in return for significant gains in performance.

4.11.1 Shortening Communication Paths

Consider the steps involved in handling client P's request to access a cached file.

1. P sends a request to VFS_P.
2. VFS_P calculates the current file offset and sends a request to FSD_r.
3. FSD_r calculates a cache reference for the cached data then responds to VFS_P.
4. VFS_P accesses the block by invoking the α-Nexus shared memory API with the shared memory region ID contained in the cache reference, then VFS_P responds to P.

Clearly, P can easily perform the offset calculations done by VFS_P in step 2, and the identity of FSD_r is known at the time a file is opened. So we could eliminate VFS_P and use a different sequence of steps:

1'. P calculates the current file offset and sends a request to FSD_r.

2'. FSD_r calculates a cache reference for the cached data and responds to P.

3'. P accesses the block by invoking the α-Nexus shared memory API with the shared memory region ID contained in the cache reference.

This modified protocol involves fewer IPC messages during file accesses, at the cost of duplicating some VFS functionality (i.e., maintaining the current file offset) in the client. However, the new client code is not likely to change the TCB for client security goals, since the client already places trust in VFS_P, which implements nearly identical code. Moreover, the duplicate functionality is straightforward to implement in the client, so even a small gain in performance justifies this optimization.

As a further optimization, we can amortize the overhead of step 2' if P makes multiple accesses to the same file. When a file is first opened, P invokes FSD_r to obtain a cache reference for the first block of the file, and P caches the shared memory region ID contained within that cache reference. During subsequent accesses to the same file, P uses the previously cached shared memory region ID rather than contacting FSD_r again.

Notice, these optimizations do not actually give P additional privileges. It may seem that P gains the ability to make arbitrary changes to the current file offset or to influence the offset calculations, since these are now located within P. But in fact, P can already completely determine the output of the offsets calculations done by VFS_P—P need only

invoke the `seek` method. P can also already access shared memory regions, as it does for mmap-oriented file access. Thus P's behavior is still governed by PDAC. Even if P were to perform incorrect offset calculations or access the wrong shared memory region, the only consequence would be that P receives incorrect data or its request is rejected when it makes the system call to access the shared memory region.

4.11.2 Leveraging Fate-Sharing

Fate-sharing [Clark 1988]—which occurs when failures are not independent—creates an opportunity to improve performance without sacrificing security. If components place full trust in each other, then isolating them contributes nothing to security, and merging them eliminates overhead. One simple example in PFS is the DDRM, which executes in the kernel rather than as a process outside of the kernel. Even were they isolated from each other, the compromise of the kernel or of the DDRM could lead to the compromise of the other. So isolating these components does not increase trustworthiness.

We also leverage fate-sharing if we merge each client P with corresponding PFS component VFS_P. To achieve this, we incorporated VFS code into the standard C library used by α-Nexus programs—that change is transparent to application programmers. Eliminating the isolation boundary between P and VFS_P allows IPC calls to be replaced with more efficient local function calls. This optimization has two potential consequences: P can now cause the compromise of VFS_P, and VFS_P can now cause the compromise of P. The former is not a concern in PFS, because P holds at least as many privileges as VFS_P. We accept the later, because VFS code in PFS is quite small and simple, so it is unlikely to cause a compromise of P.

4.11.3 Relocating Guards

By changing how and when authorization checks are performed, we can change their performance overhead. One approach is to amortize some or all of the work done for checks, with a single check serving for multiple requests. A canonical example with filesystems is when access control checks are only performed when a file is first opened, rather than for every access.

PFS amortizes the cost of checks done for I/O requests that initiate DMA transfers. Rather than have the kernel check whether the process that owns a shared memory region speaks for $PolicyMgr$ before each DMA transfer, the kernel performs this check only when shared memory regions for the block cache are allocated. The use of the α-Nexus alias table abstraction is also a form of amortization, because some of the proof checking done by the kernel is performed when an alias table entry is created rather than each time it is used.

5 Filesystem Evaluation

We performed experiments to gain insight into the costs and the benefits of instantiating security principles. The results—detailed below—confirm that instantiating the security

principles brought tangible benefits to PFS but, for several benchmarks, did not result in markedly worse performance compared to more traditional designs. To the extent possible, we attribute costs revealed by our experiments to specific principles—Mutual Suspicion, Complete Mediation, Least Privilege, or Minimization of TCBs. However, the costs are not entirely separable, because the principles are not independent. Instantiating Least Privilege, for example, tends to reduce the size of TCBs. And both Mutual Suspicion and Complete Mediation can be leveraged to instantiate the other principles.

5.1 Cost of Mutual Suspicion and Complete Mediation

Mutual Suspicion makes it difficult for components to compromise each other, and Complete Mediation requires that all requests be subject to authorization. Consequently, we attribute to these principles the cost of creating and maintaining well-defined isolation boundaries and communications channels between components. We also include costs associated with having the bulk of PFS execute outside the operating system kernel, since components that execute in the kernel would not be isolated and thus can't be mutually suspicious of each other and aren't easily constrained by authorization mechanisms.

5.1.1 Impact of Mutual Suspicion and Complete Mediation on Code Size

One way we assign cost to a design principle is to examine its impact on code size—the amount of filesystem code—since a larger code body is likely harder to develop and maintain. This cost is born mainly by system developers and, to a lesser extent, system administrators. Some operating systems do not provide suitable abstractions for isolating processes from each other by default, necessitating extra code and configuration to instantiate Mutual Suspicion and Complete Mediation for components. However, the α-Nexus process isolation and IPC abstractions are designed for building trustworthy applications, so little additional code is necessary to instantiate these principles in PFS. Further, code that executes outside of the kernel is simpler in some ways than code that executes inside the kernel. Some system services, for example, are not designed to be invoked by kernel code. Thus one might expect at most a modest increase in the amount of code required to implement a filesystem with Mutual Suspicion and Complete Mediation on α-Nexus.

We counted the lines of code (LOC) that implement PFS. See Figure 3(a). Then, as a point of comparison, we examined two alternative filesystem designs, here called UFS and KFS, that support the same interface but with a different decomposition into components.

- Figure 3(b) shows code sizes for UFS, which places most filesystem code in a single component executing as a process above the α-Nexus kernel. We obtained the UFS implementation from our PFS implementation by eliminating some guards and replacing code for handling IPC messages with equivalent code using local function calls within the single filesystem process.

- Figure 3(c) shows estimates for code sizes that would result from implementing a traditional kernel-mode filesystem design, KFS. In KFS, all filesystem code executes

	LOC
PFS Kernel-mode	
Shared mem. core	1,673
Shared mem. ext.	227
DDRM core	4,310
DDRM SATA spec.	284
DDRM ext.	64
PFS User-mode	
User-mode driver lib.	49,839
SATA disk driver	28,216
FAT32 driver	7,230
VFS layer	1,915
Policy management	2,267
Miscellaneous	4,562
PFS Total	100,587

(a)

	LOC
UFS Kernel-mode	
Shared mem. core	1,673
Shared mem. ext.	41
DDRM core	4,310
DDRM SATA spec.	284
DDRM ext.	0
UFS User-mode	
User-mode driver lib.	49,839
SATA disk driver	28,029
FAT32 driver	6,442
VFS layer	771
Policy management	1,689
Miscellaneous	4,081
UFS Total LOC	97,159

(b)

	LOC
KFS Kernel-mode	
Shared mem. core	1,673
SATA disk driver	28,029
FAT32 driver	6,442
VFS layer	771
Policy management	1,689
Miscellaneous	4,081
KFS User-mode	
n/a	0
KFS Total LOC	42,685

(c)

Figure 3: Lines of code (LOC) for PFS (a) and two alternative designs, UFS (b) and KFS (c), including some related α-Nexus kernel code. KFS counts are estimates.

within the α-Nexus kernel. We did not implement the KFS design, because α-Nexus does not support executing disk drivers within the kernel.

For each design, we also counted the lines of code that implement α-Nexus shared memory and the DDRM. These kernel facilities had to be modified for PFS, as described in Section 4.7.

UFS instantiates Mutual Suspicion and Complete Mediation between the filesystem and clients (though not between different parts of UFS). KFS is also isolated from and suspicious of clients, which invoke the filesystem through system calls. But, because KFS executes within the kernel, clients must fully trust KFS. And while KFS mediates on requests from clients, clients do not mediate on messages from KFS. Thus KFS does not instantiate Mutual Suspicion or Complete Mediation. Comparing counts for UFS and KFS in Figures 3(b) and 3(c), we found that instantiating these principles required 54,474 additional lines of code in total—more than a doubling of the code size is needed to achieve Mutual Suspicion and Complete Mediation in UFS compared to the more traditional KFS design.

By far, the largest source for the increase in UFS (and PFS) code size over KFS is due to the user-mode driver library, which contributes 49,839 lines of code. There are two reasons we might discount the cost of this library. First, this library is shared by many α-Nexus device drivers, including network and USB device drivers. Thus we can amortize its cost over many system services. Second, it not always necessary to use this library. The user-mode SATA disk driver used in UFS and PFS requires this library, but only be-

cause the driver was originally programmed against Linux's kernel-mode driver API rather than against α-Nexus's user-mode driver API.[16] Much of the user-mode driver library code (roughly 85%) provides a compatibility layer that emulates Linux's kernel-mode driver API, and that code would not be necessary had we implemented a native user-mode SATA disk driver for α-Nexus from scratch. Instead, a native α-Nexus driver would use a much smaller library containing only a few helper routines for handling interrupts, I/O requests, thread scheduling, locks and mutexes, which together account for only about 15% of the user-mode driver library, or 7,500 lines of code.

Beyond the user-mode driver library, instantiating Mutual Suspicion and Complete Mediation causes only a modest increase in code size. UFS requires 41 additional lines of kernel code so that the filesystem process is able to manage shared memory on behalf of clients. UFS also requires 4,594 LOC to implement the DDRM, which allows the kernel to mediate requests from filesystem device drivers that execute outside the kernel. Surprisingly, the DDRM was the only run-time guard required to instantiate Mutual Suspicion: for processes that communicate over IPC channels, the α-Nexus IPC abstraction suffices to prevent compromise from spreading, and no additional run-time guard is necessary. (Other run-time guards are needed to instantiate Least Privilege.)

5.1.2 Performance Implications of Mutual Suspicion and Least Privilege: Micro-benchmarks

Run-time performance is an important metric, since performance is often given by implementers as a reason to make, rather than check, assumptions. We performed a series of experiments to measure the run-time performance of certain operating system primitives, and we use the results to predict and explain the cost of instantiating principles in the design of PFS.

IPC typically imposes higher costs than system calls, but instantiating Mutual Suspicion means that clients now invoke the filesystem over IPC rather than just system calls. So PFS performance is dependent on the relative performance of the IPC versus the system call mechanisms. We implemented the following micro-benchmarks to measure their latency and throughput.

- *Syscall latency*: Perform a single null system call.
- *IPC latency*: Send a one-byte request over an IPC channel[17] and receive a one-byte response.
- *Syscall throughput*: Transfer 16 MB, in 4 KB blocks, from the kernel to a process using system calls.[18]

[16] For KFS code size estimates, we assume the size of an kernel-mode driver for α-Nexus would be comparable to the kernel-mode SATA disk driver for Linux.

[17] We use pipes to implement IPC on Linux.

[18] On Linux, reads to `/dev/zero` are used for this micro-benchmark. On α-Nexus, an equivalent system call, unrelated to the filesystem, is used. This micro-benchmark is meant to measure the speed at which kernel data

- *IPC throughput*: Transfer 16 MB, using 4 KB requests and empty responses, over an IPC channel between two processes.[19]

All experiments were performed on a 2.66 GHz Intel Core 2 platform with 3 GB of RAM, running either α-Nexus or Linux [Linux Kernel (version 2.6.29.2) 2009], as appropriate. We report the median of 100 trials for each micro-benchmark. Measured variability was low, with 80% of trials falling within $\pm 3\%$ of the median, except where noted.

On α-Nexus, system calls were measured to have substantially lower latency than IPC: 0.228 µs median syscall latency versus 5.02 µs median IPC latency. And α-Nexus system calls have higher throughput than IPC: 6,116 $\mathrm{MB/s}$ median syscall throughput versus 612 $\mathrm{MB/s}$ median IPC throughput. These results reveal a familiar trade-off between isolation and performance: IPC allows senders and recipients to be isolated from each other but has a high cost; the system call mechanism has higher performance but leaves processes vulnerable to compromises from the kernel.

For comparison, we executed the same micro-benchmarks on Linux. A similar pattern was revealed: we achieved 0.209 µs median syscall latency versus 5.09 µs median IPC latency and 6,221 $\mathrm{MB/s}$ median syscall throughput versus 701 $\mathrm{MB/s}$ median IPC throughput. The overall similarity between α-Nexus and Linux micro-benchmark performance suggests that differences between α-Nexus and Linux system call and IPC mechanisms are unlikely to account for large differences in performance for filesystems running on α-Nexus versus filesystems running on Linux.

5.1.3 Performance Implications of Mutual Suspicion and Least Privilege: Filesystem Benchmarks

To quantify the performance costs associated with Mutual Suspicion and Complete Mediation as instantiated in PFS, we measured the performance of PFS and compared it to alternate designs, including Linux FAT32, a standard Linux implementation of the FAT32 filesystem that uses a traditional kernel-mode design.

Given the micro-benchmark results above, we can predict how the performance of PFS would compare to Linux FAT32. Consider, for example, the file `open` and `close` operations. In PFS, each of these operations requires at least one IPC message (5.02 µs median latency on α-Nexus), whereas in Linux FAT32, each operation requires only a system call (0.209 µs median latency on Linux). So we should expect the median latency of `open` or `close` operations to each be about $5.02\,\mu\mathrm{s} - 0.209\,\mu\mathrm{s} = 4.81$ µs slower in PFS than in Linux FAT32.

can be written into a process's memory. So for each system call, the kernel zero-fills a message buffer specified by the micro-benchmark client, and the client does not copy or otherwise access the data returned from each system call.

[19] This micro-benchmark is meant to measure the speed at which one process's data can be copied into another process's memory over a α-Nexus IPC channel or Linux pipe. So, for each send or receive system call, the kernel copies a request or response messages from one process's memory to the other's, but neither process copies or otherwise accesses the request or response messages.

We implemented benchmarks to quantify the performance of these and other common filesystem operations, as follows.

- *Open/close latency*: Open then close a 1 MB file.
- *Enumerate latency*: Enumerate 4,096 files and 1,365 directories in a 5-level tree.
- *Read latency*: Read a single byte from an already-open 16 MB file using the streams-oriented filesystem interface.
- *Read throughput*: Read all bytes from an already-open 16 MB file using the streams-oriented filesystem interface.

These benchmarks are implemented by a filesystem client that can run in two configurations. For the *uncached configuration*, the benchmark client requests that all caches (including the filesystem's block cache and the disk's internal data cache) be emptied before each trial. For the *cached configuration*, the benchmark client warms the caches before each trial so that all requests are satisfied by the filesystem's block cache and no disk I/O is performed. We use the same machine as for the above micro-benchmarks, and we again report the median of 100 trials for each experiment. Experiments used a single 160 GB, 7,200 RPM SATA disk with a FAT32-formatted partition. According to manufacturer specifications, the disk hardware can achieve 78 $\mathrm{MB/s}$ sustained read throughput and has an average seek latency of 4.16 ms.

Results for the cached configuration of the open/close latency benchmark are as follows.[20]

	Cached Open/Close Latency (µs)		
	low	median	high
PFS	17.0	17.4	17.8
Linux FAT32	4.1	5.6	6.2

No disk I/O is performed in the cached configuration, so these results are not sensitive to disk seek latency or disk throughput. PFS achieves a median cached open/close latency of 17.4 µs, approximately $3x$ worse than the 5.6 µs median latency for Linux FAT32. We can attribute much of this noticeable performance cost to instantiation of Mutual Suspicion, since the cost of replacing two Linux system calls with two α-Nexus IPC invocations—about $2 * 4.81$ µs $= 9.62$ µs—accounts for most of the 17.4 µs $- 5.6$ µs $= 11.8$ µs difference between PFS and Linux FAT32 performance for this benchmark.

The nearly 12 µs additional latency caused by instantiating Mutual Suspicion can be significant for low-latency filesystem operations. But for filesystem operations with high latency but few IPC calls—e.g., those that are constrained by disk performance—the added latency for supporting Mutual Suspicion is negligible. The uncached configuration of the enumerate latency benchmark is one such case.

[20] Here and elsewhere we report the median result for each system tested. We also show the 10^{th} and 90^{th} percentile results—labeled low and high, respectively—to indicate the range in which the middle 80% of results fall.

	Uncached Enumerate Latency (s)		
	low	median	high
PFS	3.35	3.41	3.42
Linux FAT32	3.19	3.27	3.27

Here, median latency for PFS and Linux FAT32 differ by 0.14 s, or about 4%, with 3.41 s for PFS and 3.27 s for Linux FAT32. Performance here is likely dominated by the cost of accessing the disk, so any overhead due to IPC calls in PFS is small by comparison.

Results for uncached read latency and read throughput benchmarks reveal a similar pattern.

	Uncached Read Latency (ms)			Uncached Read Throughput (MB/s)		
	low	median	high	low	median	high
PFS	3.67	6.67	8.88	45.6	46.1	46.5
Linux FAT32	2.56	6.29	9.75	44.6	45.1	45.6

On both benchmarks, the median results for PFS and Linux FAT32 differ by less than 6%. This similarity in performance between the two systems is to be expected since, like the uncached enumerate latency benchmark, uncached read performance for both PFS and for Linux FAT32 is constrained largely by disk performance.[21]

PFS and Linux FAT32 both exhibit high variability on the uncached read latency benchmark—the slowest 10% of reads for Linux FAT32 take longer than 9.75 ms while the fastest 10% take less than 2.56 ms. We attribute this high variability to variability in disk seek latency, which should be distributed uniformly (assuming random seeks) between about 0 ms and $2 * 4.16 \text{ ms} = 8.32$ ms, given the 4.16 ms average seek latency claimed by the disk manufacturer. The slightly higher variability in Linux performance for this benchmark could be due to scheduling and lock contention between the filesystem and background processes, drivers, and interrupts. α-Nexus is a research prototype and, as such, executes few background activities that cause contention.

Cached read performance is not constrained by disk performance, so even a small per-operation run-time overheads add up quickly. One might expect Mutual Suspicion and Complete Mediation to add a significant cost for these operations in PFS. Yet this does not appear to be the case, as can be seen in the results for the cached configuration of the read latency and read throughput benchmarks.[22]

	Cached Read Latency (µs)			Cached Read Throughput (MB/s)		
	low	median	high	low	median	high
PFS	0.925	0.966	1.05	3550	3560	3570
Linux FAT32	1.40	2.70	2.78	3420	3430	3500

[21] Neither PFS nor Linux FAT32 appears able to achieve the 78 MB/s sustained read throughput that is claimed by the disk manufacturer. We did not investigate this discrepancy further.

[22] The experiment discussed here is for a benchmark client that uses the streams-oriented filesystem interface. We also performed experiments using the mmap-oriented filesystem interface. Compared to the streams-oriented client, the mmap client observed higher up-front costs to create memory mappings and lower per-access costs to access data, both on α-Nexus and on Linux. These results are unsurprising and are omitted.

PFS outperforms Linux FAT32 for cached reads—PFS achieves a median cached read latency of 0.996 µs versus 2.70 µs for Linux FAT32. However, latency for Linux FAT32 shows high variability, with 10% of trials measuring 1.40 µs or less, a value much closer to the median cached read latency in PFS (0.996 µs). By contrast, median cached read throughput for PFS and Linux FAT32 differ by less than 4%—3,560 $\mathrm{MB/s}$ median throughput for PFS versus 3,430 $\mathrm{MB/s}$ for Linux FAT32—with low variability. These results can be explained by considering how file reads are implemented in the two filesystems. Specifically, file reads in both PFS and in Linux FAT32 involve only system calls and not IPC. A Linux client invokes the VFS layer, which resides in the Linux kernel, using the `read` system call. An α-Nexus client P reads a file in PFS by invoking VFS_P using a local function call, since VFS_P resides within client P's address space, and VFS_P in turn accesses the PFS block cache using the `shm_read` system call. We suspect that for both benchmarks, filesystems are constrained only by the ability of the α-Nexus and Linux system call mechanisms to perform low latency and high throughput data transfers between the kernel and the benchmark client.[23] By taking advantage of α-Nexus shared memory and avoiding IPC in the critical path, PFS achieves similar read performance as Linux FAT32—both for cached and uncached reads—despite instantiating Mutual Suspicion.

We conclude from the results of these filesystem benchmarks that, while the cost of instantiating Mutual Suspicion and Complete Mediation can be high, costs are not borne equally by all filesystem operations and can be avoided entirely for some common operations.

5.2 Cost of Least Privilege

Instantiating Least Privilege in PFS was a matter of limiting the privileges at components and decomposing the system into fine-grained components that can then be granted correspondingly fewer privileges. The fine-grained components must instantiate Mutual Suspicion and Complete Mediation, for which there might be a cost, as described in Section 5.1. Also, guards that instantiate Least Privilege might distinguish more and finer-grained privileges, leading to added code or added run-time overhead.

5.2.1 Impact of Least Privilege on Code Size

We quantify the additional code needed to instantiate Least Privilege in PFS by comparing the code base of PFS with that of UFS, which executes outside the kernel but as a single component with broad privileges. Implementing PFS involved 100,587 total lines of code,

[23] Data is read from the filesystem using 4 KB transfers. An upper bound on cached read throughput for this message size can be obtained by considering the system call throughput micro-benchmark, discussed in Section 5.1.2. In that micro-benchmark, α-Nexus and Linux both achieve a median throughput of about 6,000 $\mathrm{MB/s}$ for 4 KB transfers. That result is higher than achieved for the cached configuration of the read throughput benchmark, but it does not include the cost of copying data from the filesystem block cache into a process's address space. Instead, it only measures the performance for the kernel to zero-fill 4 KB message buffers in response to system calls.

from Figure 3(a), versus 97,159 total lines of code for UFS, from Figure 3(b). The 3,428 additional lines of code for PFS is spread over many components. The DDRM was extended for PFS with additional code to implement Least Privilege for disk device drivers. Specifically, drivers for PFS retain privileges sufficient to initiate DMA certain transfers but are no longer granted privileges to initiate arbitrary DMA transfers. The Policy Manager component was modified to enforce a finer-grained, pervasive DAC policy, rather than simply granting all filesystem components privileges to access all filesystem meta-data stored on disk. And most PFS components were modified to include a small amount of code for creating IPC channels, for sending and receiving requests sent over IPC channels, and for guards to authenticate those requests. This diffuse impact for instantiating Least Privilege is quite different from instantiation of Mutual Suspicion and Complete Mediation, where nearly all additional code was due to just two components, as described previously in Section 5.1.1.

In retrospect, the amount of code—no more than 3,428 lines—needed in PFS to implement guards for Least Privilege is surprisingly small. By contrast, the DDRM alone is larger (4,594 LOC, from Figure 3(b)) than all of the other PFS guards combined. There are two reasons the DDRM is so much larger than other PFS guards. First, the DDRM enforces policies for a wide range of device driver requests—interrupts, I/O requests, etc.—and the policies the DDRM enforces are stateful, whereas the guards that check requests between PFS components tend to be simple and stateless. Second, the remaining PFS guards are able to leverage higher-level services not available within the kernel, such as the NAL guard library and authentication primitives that are part of the α-Nexus IPC mechanism.

5.2.2 Performance Implications of Least Privilege

There are two sources of run-time overhead for instantiating Least Privilege: one stemming from adding more and finer-grained guards, and one stemming from decomposing the system into fine-grained components. There is reason to expect the performance impact of guards to be relatively small. As discussed in Section 5.1.3, the DDRM has only modest performance implications because many disk operations have such high latency, despite the DDRM being fairly complex and stateful. PFS extensions to the DDRM for Least Privilege were minor, so we would not expect that conclusion to change. And because the remaining PFS guards are simple and stateless, we expected these guards to have a smaller performance impact than the DDRM. In the later case, components that previously had fast and unrestricted local access to data must now request access to that data using relatively expensive IPC channels or system calls.

Contrary to these expectations, our experiments indicate that the cost of enforcing some Least Privilege policies can be significant. Specifically, enforcing PDAC can be expensive because it requires access to policy meta-data stored on disk. This can be seen in the results of the open/close latency benchmark for the uncached benchmark configuration.

	Uncached Open/Close Latency (ms)		
	low	median	high
PFS	45.1	48.5	51.2
Linux FAT32	4.9	8.1	10.5

On this benchmark, PFS achieves a median latency of 48.5 ms, which is approximately $6x$ worse than the 8.1 ms median latency achieved by Linux FAT32. This discrepancy is larger than can be explained by the different costs of IPC versus system calls. We conjecture that the difference is due to the difference in how PFS and Linux FAT32 enforce DAC. Linux FAT32 enforces DAC only for the contents of files and directories. And because of limitations of the FAT32 filesystem, Linux FAT32 does not retrieve policy meta-data from the disk. Instead, Linux FAT32 uses a single pre-configured owner and ACL for all files and directories. By contrast, the PDAC policy enforced by PFS distinguishes each file, directory, or other data structure stored on disk as a separate object with a different ACL and owner. Thus, for every block accessed by PFS's FAT32 filesystem driver, FSD_r, PFS's $PolicyMgr$ requires several additional accesses to retrieve that block's owner and ACL from the `policy_table` stored on disk. There is little cache locality in these accesses for a single benchmark trial, so the cost of these disk accesses adds up.

We performed a simple analysis and an experiment to test our conjecture that PDAC enforcement accounts for the observed differences in performance of PFS and Linux FAT32 for the uncached configuration of the open/close latency benchmark. First, we counted how many disk access requests were initiated by PFS and by Linux FAT32. For PFS we counted 12 accesses for each uncached trial, 10 of which were initiated by $PolicyMgr$ and 2 of which were initiated by FSD_r. For Linux FAT32 we counted only 2 accesses for each uncached trial. If disk accesses are random, and if the disk's seek latency dominates performance costs in the uncached open/close latency benchmark, then, based on the 4.16 ms average seek latency claimed by the disk manufacturer, we should expect PFS to require about $12 * 4.16 \text{ ms} = 49.92 \text{ ms}$ and Linux FAT32 to require about $2 * 4.16 \text{ ms} = 8.32 \text{ ms}$. These numbers are close to the median observation (48.5 ms and 8.1 ms for PFS and Linux FAT32, respectively, for the uncached open/close latency benchmark). This suggests that the difference in performance measured indeed can be attributed to the different granularities at which the two implementations enforce DAC. It also suggests that PFS performance could benefit by reducing the number of disk accesses it requires to retrieve policy meta-data.

As further confirmation, we performed an experiment using a modified version of PFS that instantiates Least Privilege less extensively. We replaced $PolicyMgr$ with a different implementation, $ConstPolicyMgr$. $ConstPolicyMgr$ performs no disk accesses. Instead, it assumes a single pre-configured owner and ACL for every block, thereby enforcing a DAC policy similar to what is enforced by Linux FAT32. The modified version of PFS achieved 8.6 ms median latency for the uncached configuration of the open/close latency benchmark, a performance result that is much closer to Linux FAT32 (8.1 ms median latency for the uncached open/close latency benchmark) than to PFS (48.5 ms median latency on the that benchmark). This measurement again supports our conjecture about the differ-

ence in uncached open/close latency benchmark performance observed between PFS and Linux FAT32.

The above experiment shows the impact of enforcing PDAC for clients that access filesystem data. But Least Privilege also changes how PFS components access meta-data as well. In particular, Least Privilege required that filesystem meta-data is not stored within filesystem components, but within shared memory regions where access to the data can be mediated. Filesystem operations that require access to large amounts of meta-data can suffer because local accesses made by Linux FAT32 are replaced by system calls or shared memory accesses initiated by PFS on α-Nexus.

The impact of decomposing the filesystem in this manner can be seen in the results for the cached configuration of the enumerate latency benchmark.

	Cached Enumerate Latency (ms)		
	low	median	high
PFS	125	125	126
PFSmmap	59	59	59
Linux FAT32	36	43	43

Each trial of this benchmark causes filesystem components to perform many access to meta-data—over one thousand directories are enumerated, and each directory operation requires several access to meta-data stored in the block cache. All data is cached, so the disk's high latency does not serves to mask the cost of enforcing PDAC in this case. Instead, PFS exhibits approximately $3.5x$ worse performance compared to Linux FAT32 for the cached configuration of the enumerate latency benchmark, with PFS achieving a median latency of $125\ \mathrm{ms}$ versus a median latency of $43\ \mathrm{ms}$ for Linux FAT32.

One way to lower the performance cost of enforcing PDAC for meta-data is to amortize costs over many accesses. When implementing PFS components that access meta-data in the block cache, we had to chose either the streams-oriented interface or the mmap-oriented interface. For the PFS FAT32 filesystem driver used in the above experiments, we chose to use the streams-oriented interface, because we didn't expect this component to access any single piece of meta-data frequently enough to justify the cost of creating virtual memory mappings.

We implemented and tested a second version of PFS, called PFSmmap, in which the FAT32 filesystem driver uses the mmap-oriented interface to the block cache. In this design, the cost of creating a virtual memory mapping for each piece of meta-data is amortized over all accesses to that meta-data. Since PFSmmap amortizes the cost of creating virtual mappings, the overhead for PFSmmap to access meta-data in the block cache should be similar to direct access. This design avoids the cost of using system calls to the block cache, and it avoids the cost of enforcing PDAC on every access. It is not surprising then that the cached enumerate latency for PFSmmap is lower than for PFS: $59\ \mathrm{ms}$ median latency for PFSmmap versus $125\ \mathrm{ms}$ for PFS. And the median latency for PFSmmap ($59\ \mathrm{ms}$) is much closer to the median latency for Linux FAT32 ($43\ \mathrm{ms}$) than for PFS ($125\ \mathrm{ms}$). These results suggest that the cost of Least Privilege, in the form of an increased cost for the FAT32

filesystem driver to access meta-data stored outside of the component versus direct access to meta-data stored locally, is the largest contribution to the $3.5x$ performance differences between PFS and Linux FAT32 in the cached enumerate latency benchmark. The remaining discrepancy for this benchmark is likely due to Mutual Suspicion—specifically, the cost of invoking the filesystem over IPC channels rather than through system calls.

The differences in performance of PFS and PFSmmap highlight the trade-off discussed in Section 5.1.2: the choice of communications mechanisms—here, syscalls versus memory mapping—can affect the cost of decomposing into fine-grained components, and the costs can sometimes be avoided entirely. In hindsight, using the mmap-oriented interface within the FAT32 filesystem driver component to access meta-data, as done for PFSmmap, might have been a better implementation choice for PFS. However, we can't reach a firm conclusion without a realistic model for client file and directory access patterns.

5.3 Cost of Minimization of Trusted Computing Bases

To quantify the cost of Minimization of Trusted Computing Bases, we revisit one particular PFS design decision: merging the VFS layer into filesystem clients in an effort to gain performance, but at the cost of a potentially larger TCB for some client security goals. As described in Section 4.11.2, each VFS component of PFS, VFS_P, executes within the address space of the corresponding client P instead of as an isolated process, and P invokes VFS_P using local function calls rather than IPC. We justified this optimization by arguing that the performance cost incurred by using IPC outweighs the benefits of reducing the TCB base for security goals relating to P. We implemented a version of PFS without this optimization, leading to a larger TCB. We then used the four filesystem benchmarks—open/close latency, enumerate latency, read latency, and read throughput—to compare the performance of the default, larger TCB version (optimized for performance) and smaller TCB version (not optimized for performance).

For the uncached configuration of the filesystem benchmarks, we observed only minor differences in performance between the larger and smaller TCB versions of PFS. This was expected, since here, both versions of PFS are constrained largely by disk performance, and the performance costs incurred for disk access far outweigh the overhead of IPC above local function calls.

For the cached configuration of some filesystem benchmarks, minimizing the TCB in this manner caused a noticeable performance degradation. For the cached configuration of the open/close benchmark, we observed an increase of 10.2 µs in the median latency achieved by the smaller TCB version as compared to the larger TCB version, nearly doubling the median latency. We can attribute this increase to the overhead of IPC, because the increase is only slightly larger than twice the 5.02 µs median α-Nexus IPC latency, from Section 5.1.2, and since the smaller TCB version requires two additional IPC invocations—one IPC invocation from P to VFS_P for `open`, and a second for `close`—that were not present in the larger TCB version.

A similar analysis holds for the cached configuration of the enumerate latency benchmark, where we also observed a performance degradation for the smaller TCB version

compared to the larger TCB version of PFS.

For the cached configurations of the read latency and read throughput benchmarks, we expected the smaller TCB version of PFS to perform poorly, since each access to file contents now involves an IPC invocation from P to VFS_P versus a local function call in the larger TCB version. Moreover, in the smaller TCB version, file data is transferred through both an IPC channel and system calls, versus only system calls for the larger TCB version of PFS. Results for the cached configurations of the read latency and read throughput benchmarks using the larger TCB version were 5.20 µs and 530 $\mathrm{MB/s}$, respectively, in the median case. This is substantially worse than the performance of the larger TCB version of PFS for the same benchmarks (0.966 µs median latency and 3,560 $\mathrm{MB/s}$ median throughput, from Section 5.1.3).

The performance impact of minimizing the TCB here can be explained by considering the results from the micro-benchmarks discussed in Section 5.1.2. For the cached configuration of the read latency benchmark, the median latency (5.20 µs) is only 4% worse than the 5.02 µs median α-Nexus IPC latency (from Section 5.1.2). Similarly, for the cached configuration of the read throughput benchmark, the median throughput (530 $\mathrm{MB/s}$) is about 15% worse than the 612 $\mathrm{MB/s}$ median α-Nexus IPC throughput (from Section 5.1.2) for 4 KB messages.

The magnitude of the performance degradation we observed when minimizing the TCB in this way suggests that our design choice—trading a larger TCB for improved performance—is justifiable for common use-cases. This is particularly true if filesystem clients have TCBs much larger than the 1,915 lines of code (from Figure 3(a)) that constitute the VFS layer for PFS.

5.4 Benefits of Instantiating Principles

A realistic way to evaluate the benefits of any particular approach to security might be to conduct "red team" penetration testing. But this would not distinguish vulnerabilities in the implementation of PFS from those due to the underlying principles we instantiated or didn't instantiate in PFS. Yet that is the information we seek in this research. In fact, our PFS implementation—and the α-Nexus kernel on which it runs—is a research prototype written in C; it probably would not withstand even a modest red-team attack. This, however, does not shed light on the consequences of instantiating various security principles.

So we instead report here on measurements of the size of TCBs, believing these are a more useful way to evaluate whether well-built software would resist attack. Instantiation of security principles in PFS led to increased total code size. If the larger size of PFS made that system more vulnerable to compromise, then comparing performance with smaller-size systems would have been pointless. To address this concern, we calculated the contributions to the size of the TCB for PFS, UFS, and KFS relative to two design goals.

- We calculated the contribution to the TCB for enforcement of PDAC. Since enforcement of PDAC was our primary design goal, we interpret the results as a good predictor for filesystem trustworthiness.

- To measure the extent to which the operating system kernel—hence all other software running on the machine—trusts the filesystem, we calculated the contribution to the TCB for the integrity of kernel and process isolation boundaries.

In all cases, we include contributions both from filesystem code and from kernel code that implements PFS extensions to shared memory and the DDRM. Note that reducing the TCB for PDAC enforcement provides a benefit to all users who directly or indirectly trust PFS to correctly enforce PDAC, but reducing the TCB for kernel and process isolation benefits any user who trusts any software running on the machine. Moreover, the TCB for PDAC is a strict superset of the TCB for isolation, since enforcing PDAC also requires enforcing isolation.

PFS, UFS, KFS run on α-Nexus, which might not be representative of other operating systems. So, for comparison, we also performed the same calculations for Linux FAT32, the Linux analog to KFS on α-Nexus, and Linux FUSE, a user-mode implementation of the EXT2 filesystem based on FUSE [Szeredi 2012a] that serves as a Linux analog to UFS on α-Nexus. For Linux implementations, we calculated the contribution to the TCB for Linux's DAC policy (rather than for PDAC). We include only code relating to the filesystem and exclude the bulk of the kernel code base. We also excluded Linux disk driver code—Linux supports many such drivers, but they always execute within the Linux kernel. We thereby highlight the impact of different filesystem designs on the relative sizes of TCBs.

The following table summarizes our findings.

	α-Nexus			Linux	
	PFS (LOC) decomposed user-mode	UFS (LOC) monolithic user-mode	KFS (LOC) monolithic kernel-mode	FUSE (LOC) monolithic user-mode	FAT32 (LOC) monolithic kernel-mode
TCB contributions					
for PDAC/DAC	8,825	97,159	42,685	78,695	46,801
for isolation	6,558	6,311	42,685	46,609	46,801

These findings show that PFS contributes the fewest number of lines of code to the TCB for PDAC and nearly the fewest number of lines of code to the TCB for isolation, even though PFS has the largest code base of the three α-Nexus designs (from Figure 3). Only 6,558 LOC for PFS—about 7% of the PFS code base—is in the TCB for integrity of the kernel and other processes executing above the kernel, and only 8,825 lines of PFS code are part of the TCB for enforcing PDAC. KFS has the lowest number of lines of code (42,685 LOC, from Figure 3(c)) among the α-Nexus designs, but by locating all of this code within the kernel, KFS achieves only a moderately sized TCB for both security goals. UFS represents only a partial instantiation of the security principles. So although UFS achieves a smaller TCB than KFS for kernel and process isolation (6,311 LOC and 42,685 LOC, respectively), UFS also requires a large amount of code—97,159 lines—in the TCB for PDAC enforcement versus 42,685 LOC required by KFS.

This analysis of TCB contributions for the three α-Nexus designs indicates that pervasively instantiating security principles can reduce TCB size in comparison to a traditional

kernel-mode design or a design that only partially instantiates these security principles. Surprisingly, reducing one TCB can come at the expense of another TCB, as seen for UFS. Although we do not have a Linux filesystem analog to PFS, the TCB contributions shown above for the two Linux implementations are consistent with these conclusions. Linux FUSE—an analog of UFS—has a smaller TCB for kernel and process isolation than Linux FAT32—an analog of KFS—though only slightly (46,609 LOC and 46,801 LOC, respectively). And for DAC enforcement, Linux FUSE requires a larger amount of code—78,695 lines—in the TCB versus 46,801 LOC for Linux FAT32.[24]

One might be concerned about our choice of FAT32 as the basis for PFS. Would a more modern filesystem format like EXT3 confound our results concerning TCB contributions, because FAT32 is substantially simpler than EXT3? The same concern arises when comparing Linux FUSE (which implements EXT2, the predecessor to EXT3) and Linux FAT32. To allay such concerns, we measured the Linux EXT3 filesystem code base which, like Linux FAT32, follows a traditional kernel-mode design. We found that it contributes 53,654 LOC to each TCB. This is indeed more than the 46,801 LOC contributed by Linux FAT32. But this only means our use of FAT32 likely causes our analysis of TCB contributions to understate the case for design driven by the security principles. If we replaced PFS's FAT32 implementation with EXT3, the code base of PFS would likely be even larger than it currently is, and the TCBs for UFS and KFS would also be larger. But PFS would achieve the same small TCBs, since none of the code in PFS TCBs depends on the choice of filesystem format.

6 Related Work

6.1 File Systems

We are not the first to consider the design and implementation of a trustworthy filesystem. Halcrow [2004] provides a comprehensive overview of secure filesystems for Linux. Wright et al. [2003a] and Riedel et al. [2002] examine the performance and trustworthiness of secure filesystems, in both distributed and non-distributed settings, for a variety of operating systems. Here we discuss only a few secure filesystems that are particularly germane to our work. We organize the discussion according to the trust (or, conversely, suspicion) between filesystem components, local and remote disks or storage services, clients, and users.

Suspicion of local disks Many secure filesystems are intended to protect against theft of locally installed disks. Here, the filesystem employs encryption to prevent a disk from leaking filesystem data to an attacker who gains possession (hence, control) of the disk. One approach to implementing such a cryptographic filesystem places the filesystem code entirely within the operating system kernel (e.g., [Zadok et al. 1998; Wright et al. 2003b; Saout 2004; Halcrow 2005; Microsoft TechNet 2010]). So, all processes place full trust in

[24] If we had counted device driver code for Linux, as we did for α-Nexus, all of the Linux TCBs would appear larger by a constant amount. This does not change our conclusions.

all filesystem code, including disk device drivers and code for implementing encryption and decryption. The entire filesystem is in the TCB for every system security goal.

A more common approach moves encryption and decryption code out of the kernel by executing some or all of the filesystem as a process above the kernel (e.g., [O'Shanahan 2000; Blaze 2003; Gough 2012; Szeredi 2012b]). These filesystems can, in principle, instantiate Mutual Suspicion and Least Privilege by granting filesystem processes only privileges needed to service client requests and to perform encryption and decryption. The result would be that filesystem code—at least, the portion executing as a user-space process—can violate only those security goals that depend on that filesystem. Consequently some filesystem code need not be in all TCBs. The user-space filesystem process would not be able to violate the isolation of client processes, for instance, even if the filesystem were to become compromised. In practice, however, user-space filesystems often execute with full administrative privileges and are trusted by the kernel (hence, by all processes). Therefore they are in the TCB for every system security goal. Rather than increased trustworthiness, the motivation for moving filesystem code out of the kernel instead appears to be programming convenience, administrative convenience, and the desire to avoid accidental compromise of the kernel due to bugs in filesystem code.

Untrusted remote storage Cryptography can also be useful when disks or other storage services are accessed remotely (e.g., [Batten et al. 2001; Stein et al. 2001; Goh et al. 2003; Kallahalla et al. 2003; Li et al. 2004; Storer et al. 2009; Yao et al. 2010; Szeredi 2012b]). These filesystems encrypt data so that full trust need not be placed in a remote storage service. Consequently, the remote storage service holds few privileges beyond the ability to delete data or otherwise deny service. Filesystems that access data on a remote storage service do not rely on local disk device drivers, and in some cases (e.g., [Batten et al. 2001; Yao et al. 2010]) the filesystem is an application or library, rather than a system service. This makes Least Privilege easier to instantiate—an ordinary user can create and configure the filesystem without the cooperation of system administrators, and the resulting filesystems hold no more privileges than the user that created it. Eliminating disk device drivers could also lead to smaller TCBs.

Suspicion between filesystems and remote storage services A remote storage service provides an opportunity for both the local filesystem and the remote storage service to be suspicious of each other. Saksha [Kher and Kim 2007] instantiates Mutual Suspicion in an effort to ensure proper resource accounting and billing. Using signed transaction logs and other cryptographic techniques, Saksha ensures that neither the local filesystem nor the remote storage service holds privileges that can be used to violate the system's security goals.

Suspicion between users Using cryptography as an isolation mechanism impedes the ability to share data. For instance, CFS [Blaze 2003] uses symmetric per-user encryption keys within the filesystem. Such keys must not be revealed to processes executing on behalf

of other users, since any user that gains possession of another user's key can subsequently access any of that other user's data. This limits sharing files among users, unless the users place full trust in each other. Sharing files is also difficult if the filesystem places full trust in filesystem clients. pStore [Batten et al. 2001], for example, can't easily share files between users, because a substantial part of pStore executes as a library within each client's address space. The same problem arises in TrustedDB [Bajaj and Sion 2011], which executes the client and filesystem together as a single component on a micro-kernel or hypervisor, without any isolation between client and filesystem.

Untrusted clients In most filesystems, each client is a process that executes on behalf of some user, and a user is assumed to place full trust in such a client. Alcatraz [Liang et al. 2003] addresses the threat posed by filesystem clients that are not trustworthy, intercepting and suppressing any write request to the filesystem that was issued by an untrusted client. Solitude [Jain et al. 2008] queues write requests from a target client so that an administrator can first examine the requests and then, if desired, grant privileges to the process retroactively. These filesystems eliminate privileges that would normally be held by filesystem clients. Neither filesystem relies on encryption. VPFS [Weinhold 2006; Weinhold and Härtig 2008], by contrast, is a cryptographic filesystem that protects data stored by a single, trusted client from interference by all other clients. VPFS justifies this approach on Least Privilege grounds: no VPFS client holds privileges that can be used to access data stored by a different filesystem client.

Authorization and Authentication Secure filesystems vary substantially in their support for authorization and authentication. Much work is this area is driven by the challenges of distributed authorization spanning multiple administrative domains. Closest to our work is a filesystem by Garg and Pfenning [2010], called PCFS. That filesystem uses proof-carrying authorization [Appel and Felten 1999] to support a wide variety of authorization policies. Clients provide proofs and credentials to the filesystem guard, which enforces authorization policies specified in a formal logic. PCFS includes mechanisms for automated proof construction, credential and proof caching, and revocation. PFS guards rely on the α-Nexus alias table abstraction to achieve some of the same benefits as PCFS, though PFS lacks a general proof search procedure that can be used by filesystem clients. Miltchev et al. [2008] survey a large number of other secure distributed filesystems, both production systems and experimental prototypes, paying particular attention to the authentication and authorization mechanisms these filesystem support.

6.2 Security Principles

Our formulation of Mutual Suspicion derives from a formulation by Schroeder [1972]. That dissertation also explores the design of a filesystem in light of Mutual Suspicion. Least Privilege was originally formulated by Saltzer and Schroeder [1975] as one of eight security principles, several of which are now widely cited as fundamental principles for building se-

cure systems. The term *Complete Mediation* is often attributed to this same work by Saltzer and Schroeder [1975], though it can also be found in prior work by Price and Schell [1974]. The notion that all requests must be checked for authorization was described previously by Anderson [1972]. The term *trusted computing base* was coined by Nibaldi [1979]. Nibaldi's formulation of the security principle, which was subsequently incorporated into the Orange Book [Department of Defense 1985] and which we follow closely, builds on Saltzer and Schroeder's [1975] Principle of Economy of Mechanism, which in turn formalizes Schroeder's [1972] earlier notions of simplicity in protection mechanism design and implementation.

We based the design of PFS on an interpretation of these security principles in the context of a system comprising many isolated components, each treated as a principal that acts independently of other components, but where some components can become compromised. This is not the only possible interpretation, and there is even debate about their usefulness in practice.

All of the security principles we use date to the 1970s, yet software appears to be no more secure today than it was then. This criticism of the principles is discussed, and largely dismissed, by Smith and Marchesini [2007], who also discuss other concerns:

- The security principles are vague, easily misinterpreted, and likely impossible to fully achieve in practice.
- The security principles were developed in a very different context—commercial multi-user operating systems, in the case of Saltzer and Schroeder, and military and defense settings in which secrecy is (or was) paramount, in the case of Nibaldi. Thus one might worry that the principles address the wrong problems, present the wrong solutions, and address the wrong threats for today's pervasive, Internet-enabled, personal computing platforms.
- The security principles fail to address political, economic, and commercial realities. As a result, there are significant incentives for system designers to ignore these security principles, and few incentives to follow them.

These points have been raised by other authors as well. For instance, Viega and McGraw [2001] concede that it is easier and likely more profitable to ignore Least Privilege than it is to instantiate Least Privilege with any degree of completeness.

Mutual Suspicion Though it was formulated first, Mutual Suspicion is the least frequently cited of the three security principles we discuss here, and it is also seemingly the least controversial. The original formulation of Mutual Suspicion by Schroeder [1972] was descriptive rather than prescriptive. Our interpretation of the principle is closer to Nelson et al. [1990], who argue that each component of a system should be responsible for protecting itself by limiting reliance on information from external sources. Subsequent to that work, Woo and Lam [1992] apply Mutual Suspicion in a similar way to users in an authentication system.

Minimization of Trusted Computing Bases TCBs are often discussed in terms of a single set of system-wide security goals and a corresponding single TCB for the system. Rushby [1984], for example, examines how to minimize the TCB for embedded systems that support only a single application. A general system, even one with a single user or application, will have many security goals and, therefore, it could have many different TCBs. Trade-offs arise when, in the course of reducing the size of one TCB (e.g., by moving functionality out of one component and into another), we inadvertently enlarge the size of some other TCB.

Bernstein [2007] advocates for Minimization of TCBs as the key to building secure systems, and validates the benefits of this security principle by designing and implementing Qmail, a mail transfer agent. Qmail, which is now widely deployed, was designed with trustworthiness as an explicit goal, and it appears to have largely achieved that goal. It has an internal architecture similar to PFS: 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.

Arbaugh et al. [1997] argue that insufficient attention is paid to the lower layers of a system—hardware, firmware, BIOS, and bootstrapping code—when defining a TCB. Arbaugh et al. insist that the precise semantics of these lower layers should be fully understood and their implementations vetted, before they are included in a TCB. More recently, Smith and Marchesini [2007] reiterate this point, but also argue that these lower layers, particularly hardware components, are increasingly too complex and too poorly understood to make a suitable foundation for building a secure system.

If a TCB is interpreted as being dynamic, rather than static, then the necessity of including all lower layers in a TCB can be avoided. Flicker [McCune et al. 2008], for instance, removes the operating system kernel from the TCB for certain application-level security goals, even though most of the application runs on top of the kernel. This is accomplished using new hardware mechanisms that support a dynamic root of trust [Neiger et al. 2006; Advanced Micro Devices, Inc. 2009]: the kernel is temporarily suspended, and a small application-specified set of code is loaded into a secure, attested environment. The small piece of code thus runs directly on the lower-layer hardware and need not place full trust in the kernel.

Least Privilege Least Privilege can be instantiated for users and the privileges they hold. Motiee et al. [2010] provide empirical evidence that Least Privilege is rarely followed for Windows users. The authors of that study fault the authorization and authentication mechanisms supported by Windows for making Least Privilege impractical.

Interpreted broadly, Least Privilege applies not just to users, but to other principals as well. Much prior work concerns the application of Least Privilege to processes. Applying Least Privilege to a process typically means relying on `setuid` and similar system calls to change a process's privileges. Tsafrir et al. [2008] examine the poorly understood and poorly defined semantics of such system calls and the difficulties this raises for Least Privi-

lege. Krohn et al. [2005] investigate other ways in which common operating systems stand in the way of Least Privilege for processes.

Bernstein [2007] argues that Least Privilege—at least, its most common interpretation—is "fundamentally wrong." Here, instantiating Least Privileges is interpreted to involve two steps. First, identify the components of the system and enumerate all privileges those components hold. Second, successively remove privileges from components as long as the system still functions properly. The resulting assignment of privileges to components might then be said to instantiate Least Privilege, because each component holds only privileges that are necessary for the system to function properly. In a compelling example of a system designer being misled by Least Privilege, Bernstein discusses the case of a DNS resolver that, rather than being decomposed into several mutually suspicious components that each hold only a few privileges, was simply "Least Privilege-ized" as a single component, to little effect.

One might instead take Least Privilege as a mandate to decompose a system into fine-grained components such that security-critical privileges are held by few of the resulting components. Here, the difficulty is in choosing the appropriate decomposition and arranging for the now-isolated components to communicate with each other. There have been several attempts to simplify and automate these tasks. Programmers can rely on a library [Kilpatrick 2003] to make programming across isolation boundaries more convenient. Monolithic applications can also be decomposed into components automatically, based on program annotations [Zdancewic et al. 2001; Provos et al. 2003; Brumley and Song 2004]. Here, Least Privilege is taken to mean that the system should place full trust only in certain components, called *privileged* components, and that other, *unprivileged* components should hold few or no security-critical privileges. Thus, the automated approaches above decompose an application into two components—one privileged and one unprivileged—each executing as a processes on top of the kernel and communicating over an IPC channel. Swift [Chong et al. 2007] provides automatic decomposition for a more general case of Least Privilege, splitting an application across two or more mutually suspicious machines. Buyens et al. [2009] discuss a variety of automated and manual program-restructuring techniques for instantiating Least Privilege. These techniques include splitting large components within an application into finer-grained components and splitting coarse-grained privileges into finer-grained privileges.

Finally, Smith and Marchesini [2007] 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 Web page an inactive object that is acted upon, or is it an active subject that makes requests and may hold privileges?

References

Advanced Micro Devices, Inc. AMD I/O virtualization technology (IOMMU) specification. `http://support.amd.com/us/Processor_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf`, February 2009.

James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Vol. 2, ESD/AFSC, Hanscom AFB, Bedford, MA, October 1972. NTIS AD758206.

Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 52–62, New York, NY, November 1999. ACM Press.

William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 65–71, Los Alamitos, CA, May 1997. IEEE Computer Society.

Sumeet Bajaj and Radu Sion. TrustedDB: A trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 2011 ACM International Conference on Management of Data*, pages 205–216, New York, NY, June 2011. ACM Press.

Christopher Batten, Kenneth Barr, Arvind Saraf, and Stanley Trepetin. pStore: A secure peer-to-peer backup system. Technical Report LCS 632, Massachusetts Institute of Technology, Cambridge, MA, 2001.

Daniel J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, pages 1–10, New York, NY, November 2007. ACM Press.

Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Mark E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–283, New York, NY, December 1995. ACM Press.

Matt Blaze. A cryptographic file system for Unix. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 9–16, New York, NY, November 2003. ACM Press.

David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, pages 57–72, Berkeley, CA, August 2004. USENIX Association.

Koen Buyens, Bart de Win, and Wouter Joosen. Identifying and resolving least privilege violations in software architectures. In *Proceedings of the 4th International Conference on Availability, Reliability and Security*, pages 232–239, Los Alamitos, CA, March 2009. IEEE Computer Society. doi: 10.1109/ARES.2009.48.

Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure Web applications via automatic partitioning. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 31–44, New York, NY, October 2007. ACM Press.

Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, New York, NY, October 2001. ACM Press.

David D. Clark. The design philosophy of the DARPA Internet protocols. In *Proceedings of the 1988 ACM Conference on Communications Architectures & Protocols*, pages 106–114, New York, NY, August 1988. ACM Press. ISBN 0-89791-279-9. doi: http://doi.acm.org/10.1145/52324.52336.

Department of Defense. Department of Defense trusted computer system evaluation criteria (TCSEC). DoD 5200.28-STD, `http://csrc.nist.gov/publications/history/dod85.pdf`, December 1985.

Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison Wesley, Boston, MA, 1995.

Deepak Garg and Frank Pfenning. A proof-carrying file system. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 349–364, Los Alamitos, CA, May 2010. IEEE Computer Society.

Jeff Garzik. libATA developer's guide. `http://www.kernel.org/doc/htmldocs/libata.html`, 2012.

Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing remote untrusted storage. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, pages 131–145, Reston, VA, February 2003. Internet Society.

Valient Gough. EncFS encrypted filesystem. `http://www.arg0.net/encfs/`, 2012.

Michael Austin Halcrow. Demands, solutions, and improvements for Linux filesystem security. In *Proceedings of the 2004 Linux Symposium*, volume 1, pages 269–286, July 2004.

Michael Austin Halcrow. eCryptfs: An enterprise-class encrypted filesystem for Linux. In *Proceedings of the 2005 Linux Symposium*, volume 1, pages 201–218, July 2005.

Shvetank Jain, Fareha Shafique, Vladan Djeric, and Ashvin Goel. Application-level isolation and recovery with Solitude. In *Proceedings of the 3rd ACM EuroSys European Conference on Computer Systems*, pages 95–107, New York, NY, April 2008. ACM Press.

Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 29–42, Berkeley, CA, March 2003. USENIX Association.

Vishal Kher and Yongdae Kim. Building trust in storage outsourcing: Secure accounting of utility storage. In *Proceedings of the 26th IEEE Symposium on Reliable Distributed Systems*, pages 55–64, Los Alamitos, CA, October 2007. IEEE Computer Society.

Douglas Kilpatrick. Privman: A library for partitioning applications. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 273–284, Berkeley, CA, June 2003. USENIX Association.

Maxwell N. Krohn, Petros Efstathopoulos, Cliff Frey, M. Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Michelle Osborne, Steve Vandebogart, and David Ziegler. Make least privilege a right (not a privilege). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, Berkeley, CA, June 2005. USENIX Association.

Jinyuan Li, Maxwell N. Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating System Design & Implementation*, volume 6, pages 91–106, Berkeley, CA, December 2004. USENIX Association.

Zhenkai Liang, V. N. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. In *Proceedings of the 19th Annual Computer Security Applications Conference*, pages 182–191, Los Alamitos, CA, December 2003. IEEE Computer Society.

Linux Kernel (version 2.6.29.2). `http://www.kernel.org/`, April 2009.

Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM EuroSys European Conference on Computer Systems*, pages 315–328, New York, NY, April 2008. ACM Press.

Microsoft TechNet. BitLocker drive encryption overview. `http://technet.microsoft.com/en-us/library/cc732774.aspx`, 2010.

Stefan Miltchev, Jonathan M. Smith, Vassilis Prevelakis, Angelos Keromytis, and Sotiris Ioannidis. Decentralized access control in distributed file systems. *ACM Computing Surveys*, 40(3):10:1–10:30, August 2008. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/1380584.1380588.

Sara Motiee, Kirstie Hawkey, and Konstantin Beznosov. Do Windows users follow the principle of least privilege? Investigating user account control practices. In *Proceedings of the 6th Symposium on Usable Privacy and Security*, New York, NY, July 2010. ACM Press.

Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–178, August 2006.

Ruth Nelson, David Becker, Jennifer Brunell, and John Heimann. Mutual suspicion for network security. In *Proceedings of the 13th NIST-NCSC National Computer Security Conference*, Baltimore, MD, October 1990. National Institute of Standards and Technology.

G. H. Nibaldi. Specification of a trusted computing base (TCB). Technical Report M79-228, MITRE Corp., Bedford, MA, November 1979. NTIS ADA108831.

Declan Patrick O'Shanahan. CryptoFS: Fast cryptographic secure NFS. Master's thesis, University of Dublin, Dublin, Ireland, 2000.

William R. Price and Roger R. Schell. A secure approach to data base management system design. Technical Report MCI-74-2, ESD/AFSC, Hanscom AFB, Bedford, MA, January 1974. NTIS ADA532492.

Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, volume 12, pages 231–242, Berkeley, CA, August 2003. USENIX Association.

Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan. A framework for evaluating storage system security. In *Proceedings of the 2002 USENIX Conference on File and Storage Technologies*, pages 15–30, Berkeley, CA, January 2002. USENIX Association.

Rob Riglar. FAT16 / FAT32 file IO library. `http://hp.www.robs-projects.com/filelib.html`, 2010. Version 2.5.0.

John Rushby. A trusted computing base for embedded systems. In *Proceedings of the 7th DoD/NBS Computer Security Conference*, pages 294–311, September 1984.

Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

Christophe Saout. dm-crypt—a device-mapper crypto target. `http://www.saout.de/misc/dm-crypt/`, March 2004.

Fred B. Schneider, Kevin Walsh, and Emin Gün Sirer. Nexus authorization logic (NAL): Design rationale and applications. *ACM Transactions on Information and System Security*, 14(1):8:1–8:28, May 2011.

Michael Schroeder. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1972.

Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. Logical attestation: An authorization architecture for trustworthy computing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 249–264, New York, NY, October 2011. ACM Press.

Sean Smith and John Marchesini. *The Craft of System Security*. Addison Wesley, Boston, MA, 2007.

Christopher A. Stein, John H. Howard, and Margo I. Seltzer. Unifying file system protection. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 79–90, Berkeley, CA, June 2001. USENIX Association.

Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti. POTSHARDS—A secure, long-term storage system. *ACM Transactions on Storage*, 5(2):5:1–5:35, June 2009.

Zhendong Su and Gary Wassermann. The essence of command injection attacks in Web applications. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages*, pages 372–382, New York, NY, January 2006. ACM Press.

Miklos Szeredi. FUSE: Filesystem in user-space. `http://fuse.sourceforge.net/`, 2012a.

Miklos Szeredi. SSH filesystem. `http://fuse.sourceforge.net/sshfs.html`, 2012b.

Trusted Computing Group. Trusted platform module (TPM) specification, version 1.2. `https://www.trustedcomputinggroup.org/specs/TPM/`, March 2011.

Dan Tsafrir, Dilma Da Silva, and David Wagner. The murky issue of changing process identity: Revising "setuid demystified". *;login: The USENIX Magazine*, 33(3):56–66, June 2008.

John Viega and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison Wesley, Boston, MA, 2001.

Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, New York, NY, December 1993. ACM Press.

Carsten Weinhold. *Design and Implementation of a Trustworthy File System for L4*. PhD thesis, Technische Universität Dresden, Dresden, Germany, 2006.

Carsten Weinhold and Hermann Härtig. VPFS: Building a virtual private file system with a small trusted computing base. In *Proceedings of the 3rd ACM EuroSys European Conference on Computer Systems*, pages 81–93, New York, NY, April 2008. ACM Press.

Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. Device driver safety through a reference validation mechanism. In *Proceedings of the 8th Symposium on Operating System Design & Implementation*, pages 241–254, Berkeley, CA, December 2008. USENIX Association.

Thomas Y. C. Woo and Simon S. Lam. Authentication for distributed systems. *IEEE Computer*, 25(1):39–52, January 1992. ISSN 0018-9162. doi: 10.1109/2.108052.

Charles P. Wright, Jay Dave, and Erez Zadok. Cryptographic file systems performance: What you don't know can hurt you. In *Proceedings of the 2nd IEEE Security in Storage Workshop*, pages 47–61, Los Alamitos, CA, October 2003a. IEEE Computer Society.

Charles P. Wright, Michael C. Martino, and Erez Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the 2003 USENIX Annual Technical Conference, General Track*, pages 197–210, Berkeley, CA, June 2003b. USENIX Association.

Jinhui Yao, Shiping Chen, Surya Nepal, David Levy, and John Zic. TrustStore: Making Amazon S3 trustworthy with services composition. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud, and Grid Computing*, pages 600–605, Los Alamitos, CA, May 2010. IEEE Computer Society.

Erez Zadok, Ion Badulescu, and Alex Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-89, Computer Science Department, Columbia University, New York, NY, 1998.

Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 1–14, New York, NY, October 2001. ACM Press.