Type-Safe Disks

Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok
Stony Brook University

Appears in the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)

Abstract

We present the notion of a *type-safe disk* (TSD). Unlike a traditional disk system, a TSD is aware of the pointer relationships between disk blocks that are imposed by higher layers such as the file system. A TSD utilizes this knowledge in two key ways. First, it enables active enforcement of invariants on data access based on the pointer relationships, resulting in better security and integrity. Second, it enables semantics-aware optimizations within the disk system. Through case studies, we demonstrate the benefits of TSDs and show that a TSD presents a simple yet effective general interface to build the next generation of storage systems.

1 Introduction

Pointers are the fundamental means by which modern file systems organize raw disk data into semantically-meaningful entities such as files and directories. Pointers define three things: (1) the semantic dependency between blocks (e.g., a data block is accessible only through a pointer from an inode block); (2) the logical grouping of blocks (e.g., blocks pointed to by the same indirect block are part of the same file or directory); and even (3) the importance of a block (e.g., blocks with many outgoing pointers are important because they impact the accessibility of a large set of blocks).

Despite the rich semantic information inherently available through pointers, pointers are completely opaque to disk systems today. Due to a narrow read-write interface, storage systems view data simply as a raw sequence of uninterpreted blocks, thus losing all semantic structure imposed on the data by higher layers such as the file system or database system. This leads to the well-known *information gap* between the storage system and higher layers [8, 10]. Because of this information gap, storage systems are constrained in the range of functionality they can provide, despite the powerful processing capability and the great deal of low-level layout knowledge they have [25–27].

We propose the notion of a *type-safe disk* (TSD), a disk system that has knowledge of the pointer relationships between blocks. A TSD uses this knowledge in two key ways. First, semantic structure conveyed through pointers is used to enforce invariants on data access, providing better data integrity and security. For example, a TSD prevents access to an unallocated block. Second, a TSD can perform various semantics-aware optimizations that are difficult to provide in the current storage hierarchy [25, 26].

A TSD extends the traditional block-based read-write interface with three new primitives: block allocation, pointer creation, and pointer removal. By performing block allocation and de-allocation, a TSD frees the file system from the need for free-space management. Similar in spirit to type-safe programming languages, a TSD also exploits its pointer awareness to perform automatic garbage collection of unused blocks; blocks which have no pointers pointing to them are reclaimed automatically, thus freeing file systems of the need to track reference counts for blocks in many cases.

We demonstrate the utility of a TSD through two prototype case studies. First, we show that a TSD can provide better data security by *constraining* data access to conform to implicit trust relationships conveyed through pointers. ACCESS (A Capability Conscious Extended Storage System) is a TSD prototype that provides an independent perimeter of security by constraining data access even when the operating system is compromised due to an attack. ACCESS enforces the invariant that for a block to be accessed, a parent block pointing to this block should have been accessed in the recent past.

ACCESS also allows certain top-level blocks to be associated with explicit read and write *capabilities* (i.e., per-block keys); access to all other blocks is then validated through the *implicit* capability vested by the fact that a parent block pointing to that block was successfully accessed before. Such *path-based capabilities* enable applications to encode arbitrary operation-level access policies and sharing modes by constructing separate pointer chains for different modes of access.

Our second case study is secure delete [13], a TSD prototype that automatically overwrites deleted blocks. When the last pointer to a block is removed, our secure deletion TSD schedules the block for overwrite and will not reuse it until the overwrite completes.

Overall, we find that a TSD presents an improved division of labor between file systems and storage. By building on the existing block-based interface, a TSD requires minimal modifications to the file system. All the modifications required are *implementation-level*, unlike *design-level* modifications that are required with brand new interfaces. To demonstrate the ease with which existing file systems can be ported to TSDs, we have modified two file systems, Linux Ext2 and VFAT, to use our TSD prototype; in both cases the changes were minimal.

Despite its simplicity, we find the interface to be quite powerful, since it captures the essence of a file system's semantic structure [24]. We describe how various kinds of functionality enhancements enabled by alternative approaches [19, 27] can be readily implemented in our model. We also find that the notion of type-safety is largely independent of the exact unit of block access. For example, even with recent proposals for an object-based interface to disks [11, 19], the ability to convey relationship between objects through pointers has benefits very similar to what we illustrate in our case studies.

We evaluate our prototype implementations by using micro-benchmarks and real workloads. We find that the primary performance cost in a TSD arises from the various forms of state that the disk tracks for block allocation, capability enforcement, etc. The costs, however, are quite minimal. For typical user workloads, a TSD has an overhead of just 3% compared to regular disks.

The rest of this paper is organized as follows. In Section 2 we discuss the utility of pointer information at the disk. Section 3 discusses the design and implementation of the basic TSD framework. In Section 4 we describe file system support for TSDs. Sections 6 and 7 present detailed case studies of two applications of TSDs: ACCESS and secure deletion. We evaluate all our prototype implementations in Section 8. We discuss related work in Section 9, and conclude in Section 10.

2 Motivation

In this section we present an extended motivation.

Pointers as a proxy for data semantics. The inter-linkage between blocks conveys rich semantic information about the structure imposed on the data by higher layers. Most modern file systems and database systems make extensive use of pointers to organize disk blocks. For example, in a typical file system, directory blocks logically point to inode blocks which in turn point to indirect blocks and regular data blocks. Blocks pointed to by the same pointer block are often semantically related (e.g., they belong to the same file or directory). Pointers also define reachability: if an inode block is corrupt, the file system cannot access any of the data blocks it points to. Thus, pointers convey information about which blocks impact the availability of the file system to various degrees. Database systems are very similar in their usage of pointers. They have B-tree indexes that contain on-disk pointers, and their extent maps track the set of blocks belonging to a table or index.

In addition to being passively aware of pointer relationships, a type-safe disk takes it one step further. It actively enforces invariants on data access based on the pointer knowledge it has. This feature of a TSD enables independent verification of file system operations; more specifically, it can provide an additional perimeter of security and integrity in the case of buggy file systems or a compromised OS. As we show in Section 6, a type-safe disk can limit the damage caused to stored data, even by an attacker with root privileges. We believe this active nature of control and enforcement possible with the pointer abstraction makes it powerful compared to other more passive information-based interfaces.

Pointers thus present a simple but general way of capturing application semantics. By aligning with the core abstraction used by higher-level application designs, a TSD has the potential to enable on-disk functionality that exploits data semantics. In the next subsection, we list a few examples of new functionality (some proposed in previous work in the context of alternative approaches) that TSDs enable.

Applications. There are several possible uses of TSDs. (1) Since TSDs are capable of differentiating data and pointers, they can identify metadata blocks as those blocks that contain outgoing pointers and replicate them to a higher degree, or distribute them evenly across all the disks. This could provide graceful degradation of availability as provided by D-GRAID [26]. (2) Using the knowledge of pointers, a TSD can co-locate blocks along with their reference blocks (blocks that point to them). In general, blocks will be accessed just after their pointer blocks are accessed, and hence there would be better locality during access. (3) TSDs can perform intelligent prefetching of data because of the pointer information. When a pointer block is accessed, a TSD can prefetch the data blocks pointed to by it, and store it in the on-disk buffers for improved read performance. (4) TSDs can provide new security properties using the pointer knowledge by enforcing *implicit* capabilities. We discuss this in detail in Section 6. (5) TSDs can perform automatic secure deletion of deleted blocks by tracking block liveness using pointer knowledge. We describe this in detail in Section 7.

3 Type-Safety at the Disk Level

Having pointer information inside the disk system enables enforcement of interesting constraints on data access. For example, a TSD allows access to only those blocks that are reachable through some pointer path. TSDs manage block allocations and enforce that every block must be allocated in the context of an existing pointer path, thus preventing allocated blocks from becoming unreachable. More interestingly TSDs enable disk-level enforcement of much richer constraints for data security as described in our case study in section 6.

Enforcing such access constraints based on pointer relationships between blocks is a restricted form of *type-safety*, a well-known concept in the field of programming languages. The type information that a TSD exploits, however, is narrower in scope: TSDs just differentiate between normal data and pointers.

We now detail the TSD interface, its operation, and our prototype implementation. Figure 1 shows the architectural differences between normal disks and a TSD.

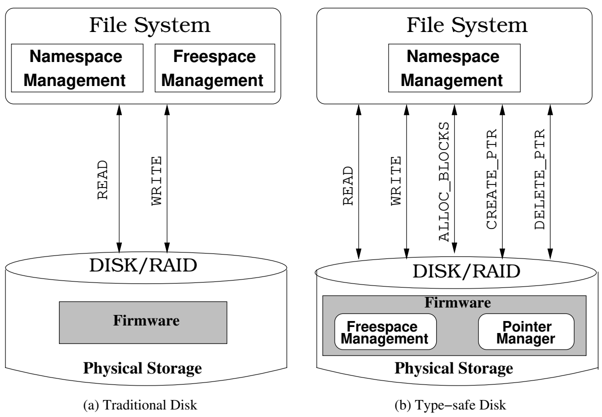


Figure 1: Comparison of traditional disks vs. type-safe disks

3.1 Disk API

A type-safe disk exports the following primitives, in addition to the basic block-based API:

- SET_BLOCKSIZE($Size$): Sets the file system block size in bytes.
- ALLOC_BLOCKS($Ref, Hint, Count$): Allocates $Count$ number of new file system blocks from the disk-maintained free block list, and creates pointers to the allocated blocks, from block Ref. Allocated blocks need not be contiguous. Ref must be a valid block number that was previously allocated. $Hint$ is the block number closest to which the new blocks should be allocated. $Hint$ can be NULL, which means the disk can choose the new block totally at its own discretion. Returns an array of addresses of the newly allocated blocks, or NULL if there are not enough free blocks on the device.
- ALLOC_CONTIG_BLOCKS($Ref, Hint, Count$): Follows the same semantics as ALLOC_BLOCKS, except that it allocates $Count$ number of contiguous blocks if available.
- CREATE_PTR($Src, Dest$): Creates a pointer from block Src to block $Dest$. Both Src and $Dest$ must be previously allocated. Returns success or failure.
- DELETE_PTR($Src, Dest$): Deletes a pointer from block Src that points to block $Dest$. Semantics similar to CREATE_PTR.
- GET_FREE: Returns the number of free blocks left.

3.2 Managing Block Pointers

A TSD needs to maintain internal data-structures to keep track of all pointers between blocks. It maintains a pointer tracking table called PTABLE that stores the set of all pointers. The PTABLE is indexed by the source block number and each table entry contains the list of destination block numbers. A new PTABLE entry is added every time a pointer is created. Based on pointer information, TSD disk blocks are classified into three kinds: (a) *Reference blocks*: blocks with both incoming and outgoing pointers (such as inode blocks). (b) *Data blocks*: blocks without any outgoing pointers but just incoming pointers. (c) *Root blocks*: a pre-determined set of blocks that contain just outgoing pointers but not incoming pointers. Root blocks are never allocated or freed, and they are statically determined by the disk. Root blocks are used for storing boot information or the primary metadata block of file systems (e.g., the Ext2 super block).

3.3 Free-Space Management

To perform free-space management at the disk level, we track live and free blocks. A TSD internally maintains an allocation bitmap, ALLOC-BITMAP, containing one bit for every logical unit of data maintained by the higher level software (e.g., a file system block). The size of a logical unit is set by the upper-level software through the SET_BLOCKSIZE disk primitive. When a new block need to be allocated, the TSD can choose a free block closest to the hint block number passed by the caller. Since the TSD can exploit the low level knowledge it has, it chooses a block number which requires the least access time from the hint block.

TSDs use the knowledge of block liveness (a block is defined to be dead if it has no incoming pointers) to perform garbage collection. Unlike traditional garbage collection systems in programming languages, garbage collection in TSD happens *synchronously* during a particular DELETE_PTR call which deletes the last incoming pointer to a block. A TSD maintains a reference count table, RTABLE, to speed up garbage collection. The reference count of a block gets incremented every time a new incoming pointer is created and is decremented during pointer deletions. When the reference count of a block drops to zero during a DELETE_PTR call, the block is marked free immediately. A TSD performs garbage collection one block at a time as apposed to performing cascading deletes. Garbage collection of reference blocks with outgoing pointers is prevented by disallowing deletion of the last pointer to a reference block before all outgoing pointers in it are deleted.

3.4 Consistency

As TSDs maintain separate pointer information, TSD pointers could become inconsistent with the file system pointers during system crashes. Therefore, upon a system crash, the consistency mechanism of the file system is triggered which checks file system pointers against TSD pointers and first fixes any inconsistencies between both. It then performs a regular scan of the file system to fix file system inconsistencies and update the TSD pointers appropriately. For example, if the consistency mechanism creates a new inode pointer to fix an inconsistency, it also calls the CREATE_PTR primitive to update the TSD internal pointers. Alternatively, we can obviate the need for consistency mechanisms by just modifying file systems to use TSD pointers instead of maintaining their own copy in their meta-data. However, this involves wide-scale modifications to the file system.

File system integrity checkers such as `fsck` for TSDs have to run in a privileged mode so that they can perform a scan of the disk without being subjected to the constraints enforced by TSDs. This privileged mode can use a special administrative interface that overrides TSD constraints and provides direct access to the TSD pointer management data-structures.

Block corruption. When a block containing TSD-maintained pointer data-structures gets corrupted the pointer information has to be recovered, as the data blocks pertaining to the pointers could still be reachable through the file system meta-data. Block corruption can be detected using well-known methods such as checksumming. Upon detection, the TSD notifies the file system, which recreates the lost pointers from its meta-data.

3.5 Prototype Implementation

We implemented a prototype TSD as a pseudo-device driver in Linux kernel 2.6.15 that stacks on top of an existing disk block driver. It contains 3,108 lines of kernel code. The TSD layer receives all block requests, and redirects the common read and write requests to the lower level device driver. The additional primitives required for operations such as block allocation and pointer management are implemented as driver `ioctl`s.

We implemented PTABLE and RTABLE as in-memory hash tables which gets written out to disk at regular intervals of time through an asynchronous commit thread. In implementing the RTABLE, we add an optimization to reduce the number of entries maintained in the hash table. We add only those blocks whose reference count is greater than one. A block which is allocated and which does not have an entry in the RTABLE is deemed to have a reference count of one and an unallocated block (as indicated by the ALLOC_BITMAP) is deemed to have a reference count of zero. This significantly reduces the size of our RTABLE, because most disk blocks have reference counts of zero or one (e.g., all data blocks have reference counts zero or one).

4 File System Support

We now describe how a file system needs to be modified to use a TSD. We first describe the general modifications required to make any file system work with a TSD. Next, we describe our modifications to two file systems, Linux Ext2 and VFAT, to use our framework.

Since TSDs perform free-space management at the disk-level, file systems using TSD are freed from the complexity of allocation algorithms, and tracking free block bitmaps and other related meta-data. However, file systems now need to call the disk API to perform allocations, pointer management, and getting the free blocks count. The following are the general modifications required to existing file systems to support type-safe disks:

1. The `mkfs` program should set the file system block size using the SET_BLOCKSIZE primitive, and store the primary meta-data block of the file system (e.g., the Ext2 super block) in one of the TSD root blocks. Note that the TSD root blocks are a designated set of well-known blocks known to the file system.
2. The free-space management sub-system should be eliminated from the file system, and TSD API should be used for block allocations. The file system routine that estimates free-space, should call the GET_FREE disk API, instead of consulting its own allocation structures.
3. Whenever file systems add new pointers to their meta-data, CREATE_PTR disk primitive should be called to create a TSD pointer. Similarly, the DELETE_PTR primitive has to be called when pointers are removed from the file system.

In the next two sub-sections we describe the modifications that we made to the Ext2 and the VFAT file systems under Linux, to support type-safe disks.

4.1 Ext2TSD

We modified the Linux Ext2 file system to support type-safe disks; we call the modified file system *Ext2TSD*. The Ext2 file system groups together a fixed number of sequential blocks into a block group and the file system is managed as a series of block groups. This is done to keep related blocks together. Each block group contains a copy of the super block, inode and block allocation data-structures, and the inode blocks. The inode table is a contiguous array of blocks in the block group that contain on-disk inodes.

To modify Ext2 to support TSDs, we removed the notion of block groups from Ext2. Since allocations and de-allocations are done by using the disk API, the file system need not group blocks based on their order. However, to perform easy inode allocation in tune with Ext2, we maintain inode groups which we call ISEGMENTS. Each isegment contains a segment descriptor that has an inode bitmap to track the number of free inodes in that isegment. The inode allocation algorithm of Ext2TSD is same as that of Ext2. The `mkfs` user program of Ext2TSD writes the super block, and allocates the inode segment descriptor blocks, and inode tables using the allocation API of the disk. It also creates pointers from the super block to all blocks containing isegment descriptors and inodes tables.

The organization of file data in Ext2TSD follows the same structure as Ext2. When a new file data or indirect block is allocated, Ext2TSD calls ALLOC_BLOCKS with the corresponding inode block or the indirect block as the reference block. While truncating a file, Ext2TSD just deletes the pointers in the indirect block branches in the right order such that all outgoing pointers from the parent block to its child blocks are deleted before deleting the incoming pointer to the parent block. Thus blocks belonging to truncated or deleted files are automatically reclaimed by the disk.

In the Ext2 file system, each directory entry contains the inode number for the corresponding file or directory. This is a logical pointer relationship between the directory block and the inode block. In our implementation of Ext2TSD, we create physical pointers between a directory block and the inode blocks corresponding to the inode numbers contained in every directory entry in the directory block. Modifying the Ext2 file system to support TSD was relatively simple. It took 8 days for us to build Ext2TSD starting from a vanilla Ext2 file system. We removed 538 lines of code from Ext2 which are mostly the code required for block allocation and bitmap management. We added 90 lines of new kernel code and modified 836 lines of existing code.

4.2 VFATTSD

The next file system we consider is VFAT, a file system with origins in Windows. Specifically, we consider the Linux implementation of VFAT. We chose to modify VFAT to support TSDs because it is sufficiently different in architecture from Ext2 and hence shows the generality of the pointer level abstraction provided by TSDs. We call our modified file system *VFATTSD*.

The VFAT file system contains an on-disk structure called the File Allocation Table (FAT). The FAT is a contiguous set of blocks in which each entry contains the logical block number of the next block of a file or a directory. To get the next block number of a file, the file system consults the FAT entries corresponding to the previous block of the file. Each file or directory's first block is stored as part of the directory entry in the corresponding directory block. The FAT entry corresponding to the last block of a file contains an EOF marker. VFAT tracks free blocks by having a special marker in the FAT entry corresponding to the blocks.

In the context of TSDs, we need not use the FAT to track free blocks. All block allocations are done using the allocation API provided by a TSD. The `mkfs` file system creation program allocates and writes the FAT blocks using the disk API. Modifying the VFAT file system to support TSDs was substantially simpler compared to Ext2, as VFAT does not manage data blocks hierarchically. We had to maintain substantially lesser number of pointers.

In VFAT, we created pointers from each directory block to all blocks belonging to files which have their directory entries in the directory block. Each FAT block points to the block numbers contained in the entries present within. The TSD therefore tracks all blocks belonging to files in the same directory block. Also, all the directory blocks and the FAT blocks contain outgoing pointers. The disk can track the set of all metadata blocks present in the file system by just checking if a block is a data block or a reference block.

Modifying the VFAT file system to support TSD was relatively straightforward. It took 4 days for us to build VFATTSD from the VFAT file system. We added 83 lines of code, modified 26 lines of code, and deleted 71 lines of code. The deleted code belonged to the free space management component of VFAT.

5 Other Usage Scenarios

In this section we discuss how the TSD abstraction that we have presented fits three other usage scenarios.

RAID systems. The TSD architecture requires a one-to-one correspondence between a file system and a TSD. However, in aggregated storage architectures such as RAID, a software or hardware layer could exist between file systems and TSDs. In this scenario, the file system gets distributed across several TSDs and no single piece has all the pointer information.

To realize the benefits of TSDs in this model, we propose the following solution: all the layers between the file systems and the TSDs should be aware of the TSD interface. Reference blocks should be replicated across the TSDs, and the software layer that performs aggregation should route the pointer management calls to the appropriate TSD that contains the corresponding data blocks. Therefore, in an aggregated storage system, each TSD contains a copy of all the reference blocks, but has only a subset of pointers pertaining to the data blocks in them. This ensures that whatever information that a single TSD has is sufficient for its own internal operations such as garbage collection, and the global structure can be used by the aggregation layer by combining the information present in each of the TSDs. The aggregation layer intercepts the CREATE_PTR and DELETE_PTR calls and invokes the disk primitives of the TSD which contains the corresponding data blocks. The aggregation layer also contains an allocation algorithm to route the ALLOC_BLOCKS call from the file system to the appropriate TSD. For example, if there are three TSDs in a software RAID system and a file is striped across the three, all the disks will contain the file's inode block. However, each disk's pointer data-structures will contain only the pointers from the inode blocks to those data blocks that are present in that disk. In this case, the first disk only contains pointers from the inode block to block offsets 0, 3, 6, and so on. A related idea explored in the context of Chunkfs is allowing files and directories to span across different file systems by having *continuation inodes* in each file system [14].

Journalling file systems. Journalling file systems maintain a persistent log of operations for easy recovery after a crash. A journalling file system that uses a TSD should pre-allocate the journal blocks using the ALLOC_BLOCKS primitive with the reference block as one of the root blocks. For example, Ext3 can create pointers from the super block to all the journal blocks. Journalling file systems should also update TSD pointers during journal recovery, using the pointer management API of the TSD, to ensure that the TSD pointer information is in sync with the file system meta-data.

Software dependent on physical locations. Software that needs to place data in the exact physical location on the disk, such as some physical backup tools, may not benefit as much from the advantages of TSDs. This is because TSDs do not provide explicit control to the upper level software to choose the precise location of a block to allocate. However, such software can be supported by TSDs by using common techniques such as preallocating all blocks in the disk and then managing them at the software level. For example, a log-structured file system can allocate all TSD blocks using the ALLOC_BLOCKS primitive during bootstrapping, and then perform its normal operation within that range of blocks.

6 Case Study: ACCESS

We describe how type-safety can enable a disk to provide better security properties than existing storage systems. We designed and implemented a secure storage system called ACCESS (***A*** *Capability* ***C****onscious* ***E****xtended* ***S****torage* ***S****ystem*) using the TSD framework; we then built a file system on top, called Ext2ACCESS.

Protecting data confidentiality and integrity during intrusions is crucial: attackers should not be able to read or write on-disk data even if they gain root privileges. One solution is to use encryption [6, 30]; this ensures that intruders cannot decipher the data they steal. However, encryption does not protect the data from being overwritten or destroyed. An alternative is to use explicit disk-level *capabilities* to control access to data [1, 11]. By enforcing capabilities independently, a disk enables an additional perimeter of security even if the OS is compromised. Others explored using disk-level versioning that never overwrites blocks, thus enabling the recovery of pre-attack data [28].

ACCESS is a type-safe disk that uses pointer information to enforce *implicit path-based* capabilities, obviating the need to maintain explicit capabilities for all blocks, yet providing similar guarantees.

ACCESS has five design goals. (1) Provide an infrastructure to limit the scope of confidentiality breaches on data stored on local disks even when the attacker has root privileges or the OS and file systems are compromised. (2) The infrastructure should also enable protection of stored data against damage even in the event of a network intruder gaining access to the raw disk interface. (3) Support efficient and easy revocation of authentication keys, which should not require costly re-encryptions upon revocation. (4) Enable applications to use the infrastructure to build strong and easy-to-use security features. (5) Support data recovery through administrative interfaces even when authentication tokens are lost.

6.1 Design

The primitive unit of storage in today's commodity disks is a fixed-size disk block. Authenticating every block access using a capability is too costly in terms of performance and usability. Therefore, there needs to be some criteria by which blocks are grouped and authenticated together. Since TSDs can differentiate between normal data and pointers, they can perform logical grouping of blocks based on the reference blocks pointing to them. For example, in Ext2 all data blocks pointed to by the same indirect block belong to the same file.

ACCESS provides the following guarantee: a block x cannot be accessed unless a valid reference block y that points to this block x is accessed. This guarantee implies that protecting access to data simply translates to protecting access to the reference blocks. Such grouping is also consistent with the fact that users often arrange files of related importance into individual folders. Therefore, in ACCESS, a single capability would be sufficient to protect a logical working set of user files. Reducing the number of capabilities required is not only more efficient, but also more convenient for users.

In ACCESS, blocks can have two capability strings: a `read` and a `write` capability (we call these *explicit capabilities*). Blocks with associated explicit capabilities, which we call *protected* blocks, can be read or written only by providing the appropriate capability. By performing an operation on a block Ref using a valid capability, the user gets an *implicit capability* to perform the same operation on all blocks pointed to by Ref, which are not directly protected (capability inheritance). If a particular reference block i points to another block j with associated explicit capabilities, then the implicit capability of i is not sufficient to access j; the explicit capability of j is needed to perform operations on it.

As all data and reference blocks are accessed using valid pointers stored on disk, root blocks are used to bootstrap the operations. In ACCESS, there are two kinds of access modes: (1) All protected blocks are accessed by providing the appropriate capability for the operation. (2) Blocks which are not protected can inherit their capability from an authenticated parent block.

ACCESS meta-data. ACCESS maintains a table named KTABLE indexed by the block number, to store the blocks' `read` and `write` capabilities. It also maintains a temporal access table called LTABLE which is indexed by the reference block number. The LTABLE has entries for all reference blocks whose associated implicit capabilities have not timed out. The timed out entries in the LTABLE are periodically purged.

Preventing replay attacks. In ACCESS, data needs to be protected even in situations where the OS is compromised. Passing clear-text capabilities through the OS interface could lead to replay attacks by a silent intruder who eavesdrops capabilities. To protect against this, ACCESS associates a sequence number with capability tokens. To read a protected block, the user has to provide a HMAC checksum of the capability (C_u) concatenated with a sequence number (S_u) ($H_u = HMAC(C_u + S_u, C_u)$). This can be generated using an external key card or a hand-held device that shares sequence numbers with the ACCESS disk system. Each user has one of these external devices, and ACCESS tracks sequence numbers for each user's external device. Upon receiving H_u for a block, ACCESS retrieves the capability token for that block from the KTABLE and computes $H_A = HMAC(C_A + S_A, C_A)$, where C_A and S_A are the capability and sequence number for the block, and are maintained by ACCESS. If H_u and H_A do not match, ACCESS denies access. Skews in sequence numbers are handled by allowing a window of valid sequence numbers at any given time.

ACCESS operation. During every reference block access, an optional timeout interval ($Interval$) can be provided, during which the implicit capabilities associated with that reference block will be active. Whenever a reference block Ref is accessed successfully, an LTABLE entry is added for it. This entry stays until $Interval$ expires. It is during this period of time, that we call the *temporal window*, all child blocks of Ref which are not protected inherit the implicit capability of accessing Ref. Once the timeout interval expires, all further accesses to the child blocks are denied. This condition should be captured by the upper level software, which should prompt the user for the capability token, and then call the disk primitive to renew the timeout interval for Ref. The value of $Interval$ can be set based on the security and convenience requirements. Long-running applications that are not interactive in nature should choose larger timeout intervals.

At any instant of time when the OS is compromised, the subset of blocks whose temporal window is active will be vulnerable to attack. This subset would be a small fraction of the entire disk data. The amount of data vulnerable during OS compromises can be reduced by choosing short timeout intervals. One can also force the timeout of the temporal window using the FORCE_TIMEOUT disk primitive described below.

ACCESS API. To design the ACCESS API, we extended the TSD API (Section 3) with capabilities, and added new primitives for managing capabilities and timeouts. Note that some of the primitives described below let the file system specify the reference block through which the implicit capability chain is established. However, as we describe later, this is only used as a hint by the disk system for performance reasons; ACCESS maintains its own structures that validate whether the specified reference block was indeed accessed, and it has a pointer to the actual block being accessed. In this section when we refer to read or write *capabilities*, we mean the HMAC of the corresponding capabilities and a sequence number.

- SET_CAPLEN($Length$): Sets the length of capability tokens. This setting is global.
- ALLOC_BLOCKS($Ref, Ref_r|C_w, Count$): Operates similar to the TSD ALLOC_BLOCKS primitive with the following two changes. (1) If Ref is protected the call takes the write capability of Ref, C_w; (2) otherwise, the call takes the reference block Ref_r of Ref, to verify that the caller has write access to Ref.
- ALLOC_CONTIG_BLOCKS($Ref, Ref_r|C_w, Count$): Same as the ALLOC_BLOCKS primitive, but allocates contiguous blocks.
- READ($Bno, Ref|C_{rw}, Timeout$): Reads the block represented by Bno. Ref is the reference block that has a pointer to Bno. C_{rw} is either the read or the write capability of block Bno. The second argument of this primitive must be Ref if Bno is not protected for read, and must be C_{rw} if Bno is protected. $Timeout$ is the timeout interval.
- WRITE($Bno, Ref|C_w, timeout$): Writes the block represented by Bno. C_w is the write capability of Bno. Other semantics are similar to READ.
- CREATE_PTR($Src, Dest, Ref_s|C_{sw}, C_{dw}|Ref_{dw}$): Creates a pointer from block Src to block $Dest$. If

Src or $Dest$ are protected, their capabilities have to be provided. For blocks which are not protected, the caller must provide valid reference blocks which point to Src and $Dest$. Note that although the pointer is created only from the source block, we need the write capability for the destination block as well; without this requirement, one can create a pointer to any arbitrary block and gain implicit write capabilities on that block.

- DELETE_PTR$(Src, Dest, Ref_s|C_{sw})$: Deletes a pointer from block Src to block $Dest$. Write credentials for Src has to be provided.
- KEY_CONTROL$(Bno, C_{ow}, C_{nr}, C_{nw}, Ref)$: This sets, unsets, or changes the read and write capabilities associated with the block Bno. C_{ow} is the old write capability of Bno. C_{nr} and C_{nw} are the new read and write capabilities respectively. A reference block Ref that has a pointer to Bno needs to be passed only while setting the write key for a block that did not have a write capability before. For all other operations, like unsetting keys or changing keys, Ref need not be specified because C_{ow} can be used for authentication.
- RENEW_CAPABILITY$(Ref, C_{rw}, Interval)$: Renews the capability for a given reference block. C_{rw} is the read or write key associated with Ref. $Interval$ is the timeout interval for the renewal.
- FORCE_TIMEOUT(Ref): Times out the implicit capabilities associated with reference block Ref.
- SET_BLOCKSIZE and GET_FREE TSD primitives (Section 3) can be called through the secure administrative interface discussed in Section 6.3.

6.2 Path-Based Capabilities

Capability systems often use capabilities at the granularity of *objects* (e.g., physical disk blocks, or memory pages); each object is associated with a capability that needs to be presented to gain access.

In contrast, the implicit capabilities used by ACCESS are *path-level*. In other words, they authenticate an access based on the path through which the access was made. This mechanism of authenticating paths instead of individual objects is quite powerful in enabling applications to encode arbitrary trust relationships in those paths. For example, a database system could have a policy of allowing any user to access a specific row in a table by doing an index lookup of a 64-bit key, but restrict scans of the entire table only to privileged users. With per-block (or per-row) capabilities, this policy cannot be enforced at the disk unless the disk is aware of the scan and index lookup operations. With path-based capabilities, the database system could simply encode this policy by constructing two separate pointer chains: one going from each block in the table to the next, and another from the index block to the corresponding table block—and just have different keys for the start of both these chains. Thus, the same on-disk data item can be differentiated for different *application-level* operations, while the disk is oblivious to these operations.

Another benefit of the path-based capability abstraction is that it enables richer modes of sharing in a file system context. Let's assume there are n users in a file system and each user shares a subset of files with another user. With traditional encryption or per-object capability systems, users has to use a separate key for each other user that shares their files; this is clearly a key management nightmare (with arbitrary sharing, we would need n^2 keys). In our model, users can use the same key regardless of how many users share pieces of their data. To enable another user to share a file, all that needs to be done is a separate link be created from the other user's directory to this specific file. The link operation needs to take capabilities of both users, but once the operation is complete, the very fact that the pointer linkage exists will enable the sharing, but at the same time limit the sharing to only those pieces of data explicitly shared.

6.3 Key Revocation and Data Recovery

ACCESS enables efficient and easy key revocation. In normal encryption based security systems, key revocation could become pretty costly in proportion to the size of the data, as all data have to be decrypted and re-encrypted with the new key. With ACCESS, one just changes the capability for the reference blocks instead of the entire set of data blocks. Data need not be modified at all while revoking capabilities.

Secure key backup is a major task in any encryption-based data protection system. Once an encryption key is lost, usually the data is fully lost and cannot be recovered. ACCESS does not have this major problem. Data is not encrypted at all, and hence even if keys are lost, data can be retrieved or the keys may be reset using the administrative interface described below.

Often system administrators need to perform backup and or administrative operations for which the restricted ACCESS interface might not be sufficient. ACCESS will have a secure administrative interface, which could be through a special hardware port requiring physical access, in combination with a master key. Using the secure administrative interface, the administrator can backup files, delete unimportant files, etc., because the data is not stored internally in encrypted format.

6.4 ACCESS Prototype

We extended our TSD prototype to implement ACCESS. We implemented additional hash tables for storing the KTABLE and LTABLE required for tracking capabilities and temporal access locality. All in-memory hash tables were periodically committed to disk through an asynchronous commit thread. The allocation and pointer management `ioctl`s in TSD were modified to take capabilities or reference blocks as additional arguments. We implemented the KEY_CONTROL primitive as a new `ioctl` in our pseudo-device driver.

To authenticate the `read` and `write` operations, we implemented a new `ioctl`, KEY_INPUT. We did this to simplify our implementation and not modify the generic block driver. The KEY_INPUT `ioctl` takes the block number and the capabilities (or reference blocks) as arguments. The upper level software should call this `ioctl` before every read or write operation to authenticate the access. Internally, the disk validates the credentials provided during the `ioctl` and stores the success or failure state of the authentication. When a read or write request is received, ACCESS checks the state of the previous KEY_INPUT for the particular block to allow or disallow access. Once access is allowed for an operation, the success state is reset. When a valid KEY_INPUT is not followed by a subsequent read or write for the block (e.g., due to software bugs), we time out the success state after a certain time interval.

6.5 The Ext2ACCESS File System

We modified the Ext2TSD file system described in Section 4.1 to support ACCESS; we call the new file system *Ext2ACCESS*. To demonstrate a usage model of ACCESS disks, we protected only the inode blocks of Ext2ACCESS with read and write capabilities. All other data blocks and indirect blocks had implicit capabilities inherited from their inode blocks. This way users can have a single read or write capability for accessing a whole file. An alternative approach may be to protect only directory inode blocks. ACCESS provides an infrastructure for implementing security at different levels, which upper level software can use as needed.

To implement per-file capabilities, we modified the Ext2 inode allocation algorithm. Ext2 stores several inodes in a single block; so in Ext2ACCESS we needed to ensure that an inode block has only those inodes that share the same capabilities. To handle this, we associated a *capability table* with every isegment (Section 4.1). The capability table persistently stores the checksums of the capabilities of every inode block in the particular isegment. Whenever a new inode needs to be allocated, an isegment with the same key is chosen if available.

Ext2ACCESS has two file system `ioctl`s, called SET_KEY and UNSET_KEY, which can be used by user processes to set and unset capabilities for files. The life of a user's key in kernel memory can be decided by the user. For example, a user can call the SET_KEY `ioctl` before an operation and then immediately call the UNSET_KEY `ioctl` after the operation is completed to erase the capability from kernel memory; in this case the life of the key in kernel memory is limited to a single operation. Ext2ACCESS uses the KEY_INPUT device `ioctl` of ACCESS to send the user's key before reading an inode block. For all other blocks, it sends the corresponding reference block as an implicit capability, for temporal authentication.

An issue that arises in Ext2ACCESS is that general file system meta-data such as super block and descriptors need to be written to all the time (and hence must have their capabilities in memory). This can potentially make them vulnerable to modifications by attackers. We address this vulnerability by mapping these blocks to root blocks and enforce that no pointer creations or deletions can be made to root blocks except through an administrative interface. Accordingly, `mkfs` creates set of pointers to the relevant inode bitmap and isegment descriptor blocks, but this cannot change after that. Thus, we ensure confidentiality and write protection of all protected user files and directories.

Although the above solution protects user data during attacks, the contents of the metadata blocks themselves could be modified (for example, free block count, inode allocation status, etc). Although most of this information can be reconstructed by querying the pointer structure from the disk, certain pieces of information are hard to reconstruct. Our current implementation does not handle this scenario, but there are various solutions to this problem. First, we could impose that the disk perform periodic snapshotting of root blocks; since these are very few in number, the overhead of snapshotting will be minimal. Alternatively, some amount of NVRAM could be used to buffer writes to these global metadata blocks and periodically (say once a day) an administrator "commits" these blocks to disk using a special capability after verifying its integrity.

7 Case Study: Secure Deletion

In this section we describe our next case study: a disk system that automatically performs secure deletion of blocks that are freed. We begin with a brief motivation and then move on to the design and implementation of our *Secure Deletion Type-Safe Disk* (SDTSD).

Data security often includes the ability to delete data such that it cannot be recovered [3, 13, 23]. Several software-level mechanisms exist today that delete disk data securely [16, 22]. However, these mechanisms are fundamentally insecure compared to disk-level mechanisms [25], because the former do not have knowledge of disk internals and therefore cannot guarantee that deleted data is overwritten.

Since a TSD automatically tracks blocks that are not used, obtaining liveness information about blocks is simple as described in Section 3. Whenever a block is garbage collected, an SDTSD just needs to securely delete the block by overwriting it one or more times. The SDTSD must also ensure that a garbage collected block that is not yet securely deleted is not re-allocated; an SDTSD achieves this by deferring the update of the ALLOC_BITMAP until a block is securely deleted.

To improve performance, an SDTSD overwrites blocks in batches. Blocks that are garbage collected are automatically added to a secure-deletion list. This list is periodically flushed and the blocks to be securely deleted are sorted for sequential access. Once a batch of blocks is overwritten multiple times, the ALLOC_BITMAP is updated to mark all those blocks as free.

We extended our prototype TSD framework described in Section 3.5 to implement secure-deletion functionality. Whenever a block is garbage collected, we add the block number to a list. An asynchronous kernel thread wakes up every second to flush the list into a buffer, sort it, and perform overwrites. The number of overwrites per block is configurable. We added 403 lines of kernel code to our existing TSD prototype.

8 Evaluation

We evaluated the performance of our prototype TSD framework in the context of Ext2TSD and VFATTSD. We also evaluated our prototype implementations of ACCESS and secure delete. We ran general-purpose workloads and also micro-benchmarks on our prototypes and compared them with unmodified Ext2 and VFAT file systems on a regular disk. This section is organized as follows: first we talk about our test platform, configurations, and procedures. Next, we analyse the performance of the TSD framework with the Ext2TSD and VFATTSD file systems. Finally, we evaluate our prototypes for ACCESS and SDTSD.

Test infrastructure. We conducted all tests on a 2.8GHz Xeon with 1GB RAM, and a 74GB, 10Krpm, Ultra-320 SCSI disk. We used Fedora Core 4, running a vanilla Linux 2.6.15 kernel. To ensure a cold cache, we unmounted all involved file systems between each test. We ran all tests at least five times and computed 95% confidence intervals for the mean elapsed, system, user, and wait times using the Student-t distribution. In each case, the half-widths of the intervals were less than 5% of the mean. Wait time is the elapsed time less CPU time used and consists mostly of I/O, but process scheduling can also affect it. We recorded disk statistics from `/proc/diskstats` for our test disk. We provide the following detailed disk-usage statistics: the number of read I/O requests (`rio`), number of write I/O requests (`wio`), number of sectors read (`rsect`), number of sectors written (`wsect`), number of read requests merged (`rmerge`), number of write requests merged (`wmerge`), total time taken for read requests (`ruse`), and the total time taken for write requests (`wuse`).

Benchmarks and configurations. We used Postmark v1.5 to generate an I/O-intensive workload. Postmark stresses the file system by performing a series of operations such as directory lookups, creations, and deletions on small files [17]. For all runs, we ran Postmark with 50,000 files and 250,000 transactions.

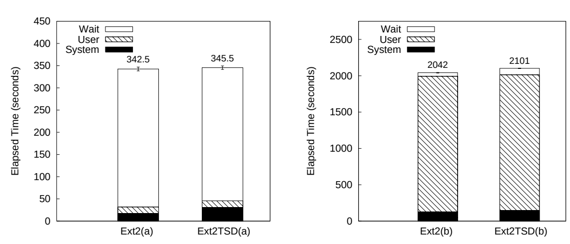
To simulate a relatively CPU-intensive user workload, we compiled the Linux kernel source code. We used a vanilla Linux 2.6.15 kernel, and analyzed the overheads of Ext2TSD and Ext2ACCESS, for the `untar`, `make oldconfig`, and `make` operations combined.

To isolate the overheads of individual file system operations, we ran micro-benchmarks that analyze the overheads associated with the `create`, `lookup`, and `unlink` operations. For all micro-benchmarks, we used a custom user program that creates 250 directories and 1,000 files in each of these directories, creating a total of 250,000 files. For performing lookups, we called the `stat` operation on each of these files. We called `stat` by specifying the full path name of the files so that `readdir` was not called. For the `unlink` micro-benchmarks, we removed all 250,000 files created.

Unless otherwise mentioned, the system time overheads were caused by the hash table lookups required during the CREATE_PTR and DELETE_PTR TSD calls. This CPU overhead is due to the fact that our prototype is implemented as a pseudo-device driver that runs on the same CPU as the file system. In a real TSD setting, the hash table lookups will be performed by the processor embedded in the disk and hence will not influence the overheads on the host system.

8.1 Ext2TSD

We analyze the overheads of Ext2TSD over our TSD framework in comparison with the overheads of regular Ext2 over a regular disk. We discuss the Postmark, kernel compilation, and micro-benchmark results.



(a) Postmark (b) Kernel compile

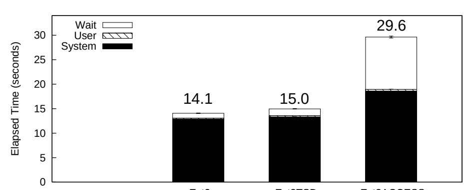
	rio	ruse	rsect	rmerge	wio	wuse	wsect	wmerge
Ext2(a)	1456	48K	14K	275	162K	43M	3M	62K
Ext2TSD(a)	721	66K	13K	927	170K	49M	3M	221K
Ext2(b)	16K	133K	771K	2K	27K	3M	3M	54K
Ext2TSD(b)	17K	105K	567K	50K	20K	2M	3M	338K

Figure 2: Postmark and Kernel compile results for Ext2TSD

Postmark. Figure 2(a) shows the comparison of Ext2TSD over TSD with regular Ext2. Ext2TSD has a system time overhead of 81% compared to regular Ext2. This is because of the hash table lookups required for creating and deleting pointers. The wait time of the Ext2TSD configurations was 5% lower than regular Ext2. This is because of better spatial locality in Ext2TSD for the Postmark workload. This is evidenced by the higher `rmerge` and `wmerge` values of Ext2TSD compared to Ext2. Ext2's allocation policy takes into account future file growth and hence leaves free blocks between newly created files. In Ext2TSD, we did not implement this policy and hence we have better locality for small files. Overall, the elapsed times for Ext2 and Ext2TSD under Postmark are similar.

Kernel compile. The Ext2TSD results for the kernel compilation benchmark are shown in Figure 2(b). The wait time overhead for Ext2TSD is 77%. This increase in wait time is not because of increase in I/O, as shown in the disk statistics. The increase is because of the increased sleep time of the Postmark process context while the TSD commit thread (described in Section 3.5) preempts it to commit the hash tables. The asynchronous commit thread runs in a different context and has to traverse all hash tables to commit them, taking more system time, which is reflected as wait time in the context of Postmark. Since a kernel compile is not an I/O-intensive workload, the system time overhead is lower than the overhead for Postmark. The elapsed time overhead of Ext2TSD compared to Ext2 under this benchmark is 3%.

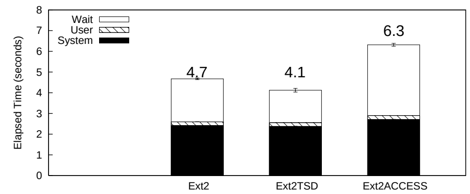
Micro-benchmarks. We ran the CREATE, LOOKUP, and UNLINK micro-benchmarks on Ext2TSD and compared them with Ext2TSD.



	rio	ruse	rsect	rmerge	wio	wuse	wsect	wmerge
Ext2	3609	14K	63k	4K	160	28K	71K	7944
Ext2TSD	3806	14K	63k	4K	156	19K	71K	8690
Ext2ACCESS	5120	20K	63k	3K	341	70K	71K	8523

Figure 3: Create micro-benchmark results

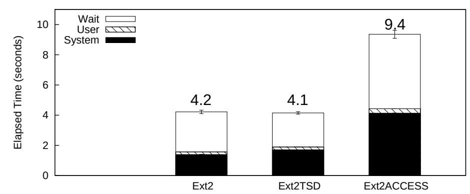
Figure 3 shows the results for the CREATE micro-benchmark. The wait time overhead is 36%, which is due to the increase in the sleep time due to CPU context switches required for the TSD commit thread. Since this benchmark has a significant system time component, this is more pronounced. The `wuse` and `ruse` values in the disk statistics have not increased and hence the higher wait time is not because of additional I/O.



	rio	ruse	rsect	rmerge	wio	wuse	wsect	wmerge
Ext2	9K	2096	71k	0	0	0	0	0
Ext2TSD	9K	1593	71k	0	0	0	0	0
Ext2ACCESS	9K	1427	71k	0	0	0	0	0

Figure 4: Lookup micro-benchmark results

Figure 4 shows the results of the LOOKUP micro-benchmark. Ext2TSD had a 12% lower elapsed time than Ext2. This is mainly because of the 24% savings in wait time thanks to the better spatial locality evidenced by the `ruse` value of the disk statistics.



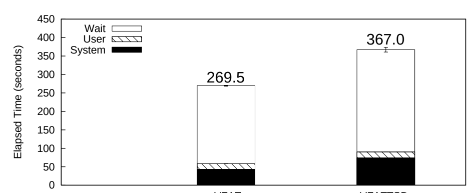
	rio	ruse	rsect	rmerge	wio	wuse	wsect	wmerge
Ext2	8850	2148	71k	0	70	17K	63K	7782
Ext2TSD	8856	1627	71k	0	74	18K	63K	7771
Ext2ACCESS	8857	1471	71k	0	236	75K	63K	7611

Figure 5: Unlink micro-benchmark results

Figure 5 shows the results of the UNLINK micro-benchmark. Ext2TSD is comparable to Ext2 in terms to elapsed time. This is because the 21% increase in system time is compensated for by the 12% decrease in the wait time, due to better spatial locality.

8.2 VFATTSD

We evaluated the overheads of VFATTSD compared to VFAT, by running Postmark on a regular VFAT file system and on VFATTSD. Figure 6 shows the Postmark results for VFATTSD. The increase in wait time in VFATTSD (31%) is due to the increased seek times while updating FAT entries. This is because VFATTSD's FAT blocks are not contiguous. This is evident from the increased value of `wuse` for similar values of `wsect`.

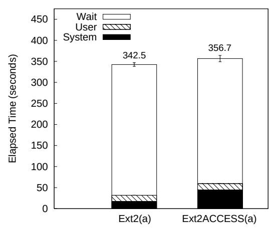


	rio	ruse	rsect	rmerge	wio	wuse	wsect	wmerge
VFAT	3601	141K	3601	0.2	199K	27M	3M	2.8M
VFATTSD	1765	42K	5884	4119	220K	50M	2.6M	2.4M

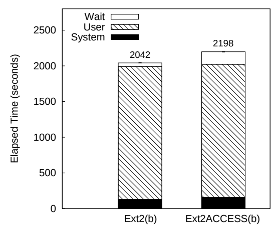
Figure 6: Postmark results for VFATTSD

8.3 ACCESS

We now discuss the results for Ext2ACCESS under Postmark, kernel compilation, and micro-benchmarks.



(a) Postmark



(b) Kernel compile

	rio	ruse	rsect	rmerge	wio	wuse	wsect	wmerge
Ext2(a)	1456	48K	14K	275	162K	43M	3M	62K
Ext2ACCESS(a)	1116	113K	19K	1241	173K	50M	3M	222K
Ext2(b)	16K	133K	771K	2K	27K	3M	3M	54K
Ext2ACCESS(b)	19K	119K	634K	60K	20K	2M	3M	338K

Figure 7: Postmark and Kernel compile results for Ext2ACCESS

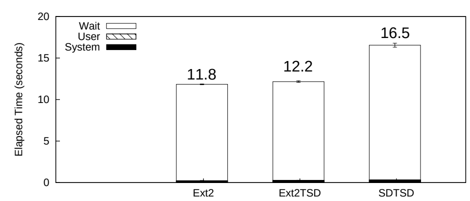
Postmark. Figure 7(a) shows the Postmark results for Ext2ACCESS and Ext2. The overheads of Ext2-ACCESS are similar to Ext2TSD except that the system time overheads have increased significantly. Ext2ACCESS has 60% more system time than Ext2. This is because of the additional information such as keys and the temporal locality of reference blocks that ACCESS needs to track. Even though ACCESS incurs the overhead to write the capability table (Section 6.5) for each isegment, it has not affected the wait time because of better spatial locality.

Kernel compile. Figure 7(b) shows the results for the Ext2ACCESS kernel compile benchmark. Ext2ACCESS had 21% more system time than Ext2, due to the additional information being tracked by ACCESS. Ext2ACCESS's wait time was 2.5 times larger because of the increase in sleep time of the compile process context due to the preemption of the TSD commit thread. The sleep time has increased compared to Ext2TSD because ACCESS is more CPU-intensive than Ext2TSD.

Micro-benchmarks. Figures 3, 4, and 5 also show the results for the CREATE, LOOKUP, and the UNLINK micro-benchmarks for Ext2ACCESS, respectively. For the CREATE workload the system time and wait time of Ext2ACCESS increased by 44% and 2.2 times, respectively, compared to regular Ext2. The increase in wait time is because of two reasons. First, the I/O time for Ext2ACCESS is greater because of the additional seeks required to access the capability table. This is shown in the disk statistics. The `wuse` and `ruse` values of Ext2ACCESS are significantly higher than regular Ext2. Second, the preemption of the benchmark process context by the TSD commit thread has resulted in increased sleep time. The results for the lookup workload are similar to that of Ext2TSD, except for the 12% increase in system time. For the unlink workload, Ext2ACCESS shows significant overheads: 90% more system time and 85% more wait time. This is because unlinking files involve several calls to the DELETE_PTR disk primitive which requires multiple hash table lookups. The increase in wait time is due to the increase in the time taken for writes as evidenced by the high `wuse` value of disk statistics. This is because of the additional seeks required to update the capability tables for each isegment.

8.4 Secure Deletion

To evaluate the performance of our next case study (SDTSD), we ran an unlink micro-benchmark. Figure 8 shows the results of this benchmark. The I/O overhead of SDTSD over Ext2TSD was 40% compared to regular Ext2, mainly because of the additional I/O caused by overwrites for secure deletion. This is evidenced by the high `wsect` and `wuse` values for SDTSD, as expected.



	rio	ruse	rsect	rmerge	wio	wuse	wsect	wmerge
Ext2	10K	2111	71k	0	70	16K	63K	7783
Ext2TSD	10K	1663	71k	0	73	18K	63K	7772
SDTSD	10K	1452	71k	0	237	77K	63K	7610

Figure 8: Unlink benchmark for secure delete

9 Related Work

Type-safety. The concept of type safety has been widely used in the context of programming languages. Type-safe languages such as Java are known to make programming easier by providing automatic memory management. More importantly, they improve security by restricting memory access to legal data structures. Type-safe languages use a philosophy very similar to our model: a capability to an encompassing data structure implies a capability to all entities enclosed within it. Type-safety has also been explored in the context of building secure operating systems. For example, the SPIN operating system [4] enabled safe kernel-level extensions by constraining them to be written in Modula-3, a type-safe language. Since the extension can only access objects it has explicit access to, it cannot change arbitrary kernel state. More recently, the Singularity operating system [15] used a similar approach, attempting to improve OS robustness and reliability by using type-safe languages and clearly defined interfaces.

Interface between file systems and disks. Our work is closely related to a large body of work examining new interfaces between file systems and storage. For example, logical disks expand the block-based interface by exposing a list-based mechanism that file systems use to convey grouping between blocks [7]. The Universal File Server [5] has two layers where the lower layer exists in the storage level, thereby conveying directory-file relationships to the storage layer. More recent research has suggested the evolution of the storage interface from the current block-based form to a higher-level abstraction. Object-based Storage Device (OSD) is one example [19]; in OSDs the disk manages variable-sized objects instead of blocks. Similar to TSD, object-based disks handle block allocation within an object, but still do not have information on the relationships across objects. Another example is Boxwood [18]; Boxwood considers making distributed file systems easier to develop by providing a distributed storage layer that exports higher-level data structures such as B-Trees. Unlike many of these interfaces, TSD considers backwards compatibility and ease of file system modification as an important goal. By following the block-based interface and augmenting it with minimal hooks, we enable file systems to be more readily portable to this interface, as this paper demonstrates. Others examine the storage interface by trying to keep the interface constant, but move some intelligence into the disk system. For example, the Loge disk controller implemented eager-writing by writing to a block closest to its disk arm [9]. The log-based programmable disk [29] extended this work, adding free-space compaction. These systems, while being easily deployable by not requiring interface change, are quite limited in the functionality they extend to disks.

A more recent example of work on improving storage functionality without changing the interface is Semantically-smart Disk Systems (SDSs) [27]. An SDS enables rich functionality by automatically tracking information about the file system or DBMS using the storage system, by carefully watching updates. However, semantic disks need to be tailored to the specifics of the file system above. In addition, they involve a fair amount of complexity to infer semantic information underneath asynchronous file systems. As the authors point out [25], SDS is valuable when the interface cannot be changed, but serves better as an evolutionary step towards an eventual change to an explicit interface such as TSD.

Capability-based access control. Network-Attached Secure Disks (NASDs) incorporate capability based access control in the context of distributed authentication using object-based storage [1, 11, 20]. Temporal timeouts in ACCESS are related to caching capabilities during a time interval in OSDs [2]. The notion of using a single capability to access a group of blocks has been explored in previous research [1, 12, 21].

In contrast to their object-level capability enforcement, ACCESS uses implicit path-based capabilities using pointer relationships between blocks.

10 Conclusions

In this paper, we have taken the well-known concept of type-safety and applied it in the context of disk storage. We have explored a simple question: what can a disk do if it knew about pointers? We find that pointer information enables rich functionality within storage, and also enables better security through active enforcement of constraints within the disk system. We believe that this pointer abstraction explores an interesting and effective design choice in the large spectrum of work on alternative interfaces to storage.

Our experience with TSDs and the case studies has also pointed to some limitations with this approach. First, TSDs assume that a block is an atomic unit of file system structure. This assumption makes it hard to enforce constraints on data objects that occupy a partial block (e.g., multiple inodes per block). Second, the lack of higher level control over block allocation may limit the benefits of TSDs with software that need to place data in the exact physical locations on disk. While the current interface presents a reasonable choice, only future research will identify if more fine tuning is required.

11 Acknowledgments

We like to thank the anonymous reviewers for their helpful comments, and especially our shepherd Garth Gibson, whose meticulous and detailed comments helped improve the work. We thank Muthian Sivathanu for his valuable feedback during the various stages of this project. We would also like to thank the following people for their comments and suggestions on the work: Remzi H. Arpaci-Dusseau, Tzi-cker Chiueh, Christos Karamanolis, Patrick McDaniel, Ethan Miller, Abhishek Rai, R. Sekar, Radu Sion, Charles P. Wright, and the members of our research group (File systems and Storage Lab at Stony Brook).

This work was partially made possible by NSF CAREER EIA-0133589 and NSF CCR-0310493 awards.

References

[1] M. K. Aguilera, M. Ji, M. Lillibridge, J. MacCormick, E. Oertli, D. Andersen, M. Burrows, T. Mann, and C. A. Thekkath. Block-Level Security for Network-Attached Disks. In *Proc. of the Second USENIX Conf. on File and Storage Technologies*, pp. 159–174, San Francisco, CA, March 2003.

[2] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and

L. Yerushalmi. Towards an object store. In *Mass Storage Systems and Technologies (MSST)*, 2003.

[3] S. Bauer and N. B. Priyantha. Secure Data Deletion for Linux File Systems. In *Proc. of the 10th Usenix Security Symposium*, pp. 153–164, Washington, DC, August 2001.

[4] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proc. of the 15th ACM Symposium on Operating System Principles*, pp. 267–284, Copper Mountain Resort, CO, December 1995.

[5] A. D. Birrell and R. M. Needham. A universal file server. In *IEEE Transactions on Software Engineering*, volume SE-6, pp. 450–453, September 1980.

[6] M. Blaze. A Cryptographic File System for Unix. In *Proc. of the first ACM Conf. on Computer and Communications Security*, pp. 9–16, Fairfax, VA, 1993

[7] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: A new approach to improving file systems. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003.

[8] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. In *Proc. of the Annual USENIX Technical Conf.*, pp. 177–190, Monterey, CA, June 2002.

[9] R. English and A. Stepanov. Loge : A self-organizing disk controller. *HP Labs, Tech. Rep.*, HPL91(179), 1991.

[10] G. R. Ganger. Blurring the Line Between OSes and Storage Devices. Tech. Rep. CMU-CS-01-166, CMU, December 2001.

[11] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proc. of the Eighth International Conf. on Architectural Support for Programming Langauges and Operating Systems (ASPLOS-VIII)*, pp. 92–103, New York, NY, December 1998

[12] H. Gobioff. *Security for a High Performance Commodity Storage Subsystem*. PhD thesis, Carnegie Mellon University, May 1999. citeseer.ist.psu.edu/article/gobioff99security.html.

[13] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *Proc. of the Sixth USENIX UNIX Security Symposium*, pp. 77–90, San Jose, CA, July 1996.

[14] V. Henson. Chunkfs and continuation inodes. *The 2006 Linux Filesystems Workshop (Part III)*, 2006.

[15] G. Hunt, J. Laurus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An Overview of the Singularity Project. Tech. Rep. MSR-TR-2005-135, Microsoft Research, 2005.

[16] N. Joukov and E. Zadok. Adding Secure Deletion to Your Favorite File System. In *Proc. of the third international IEEE Security In Storage Workshop*, San Fransisco, CA, December 2005.

[17] J. Katcher. PostMark: A New Filesystem Benchmark. Tech. Rep. TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[18] J. MacCormick, N. Murphy, M. Najork, C. Thekkath, and L. Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proc. of the 6th Symposium on Operating Systems Design and Implementation*, pp. 105–120, San Francisco, CA, December 2004.

[19] M. Mesnier, G. R. Ganger, and E. Riedel. Object based storage. *IEEE Communications Magazine*, 41, August 2003. ieeexplore.ieee.org.

[20] E. Miller, W. Freeman, D. Long, and B. Reed. Strong Security for Network-Attached Storage. In *Proc. of the First USENIX Conf. on File and Storage Technologies*, pp. 1–13, Monterey, CA, January 2002.

[21] E. L. Miller, W. E. Freeman, D. D. E. Long, and B. C. Reed. Strong security for network-attached storage. In *USENIX Conf. on File and Storage Technologies (FAST)*, pp. 1–14, jan 2002.

[22] *Overwrite, Secure Deletion Software*. `www.kyuzz.org/antirez/overwrite`.

[23] R. Perlman. Secure Deletion of Data. In *Proc. of the third international IEEE Security In Storage Workshop*, San Fransisco, CA, December 2005.

[24] M. Sivathanu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S Jha. A Logic of File Systems. In *Proc. of the Fourth USENIX Conf. on File and Storage Technologies*, pp. 1–16, San Francisco, CA, December 2005.

[25] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or Death at Block-Level. In *Proc. of the 6th Symposium on Operating Systems Design and Implementation*, pp. 379–394, San Francisco, CA, December 2004.

[26] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proc. of the Third USENIX Conf. on File and Storage Technologies*, pp. 15–30, San Francisco, CA, March/April 2004.

[27] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proc. of the Second USENIX Conf. on File and Storage Technologies*, pp. 73–88, San Francisco, CA, March 2003.

[28] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *Proc. of the 4th Usenix Symposium on Operating System Design and Implementation*, pp. 165–180, San Diego, CA, October 2000.

[29] R. Y. Wang, T. E. Anderson, and D. A. Patterson. Virtual Log Based File Systems for a Programmable Disk. In *Proc. of the Third Symposium on Operating Systems Design and Implementation*, pp. 29–44, New Orleans, LA, February 1999.

[30] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proc. of the Annual USENIX Technical Conf.*, pp. 197–210, San Antonio, TX, June 2003.