TrueErase: Per-File Secure Deletion for the Storage Data Path

Sarah Diesburg, Christopher Meyers, Mark Stanovich, Michael Mitchell, Justin Marshall, Julia Gould, and An-I Andy Wang
Florida State University
{diesburg, meyers, stanovic, mitchell, jmarshal, gould, awang}@cs.fsu.edu

Geoff Kuenning
Harvey Mudd College
geoff@cs.hmc.edu

ABSTRACT

The ability to securely delete sensitive data from electronic storage is becoming important. However, current per-file deletion solutions tend to be limited to a segment of the operating system's storage data path or specific to particular file systems or storage media.

This paper introduces TrueErase, a holistic secure-deletion framework. Through its design, implementation, verification, and evaluation, TrueErase shows that it is possible to build a legacy-compatible full-storage-data-path framework that performs per-file secure deletion and works with common file systems and solid-state storage, while handling common system failures. In addition, this framework can serve as a building block for encryption- and tainting-based secure-deletion systems.

Categories and Subject Descriptors

D.4.2 [**Storage Management**]: Allocation/deallocation strategies, D.4.3 [**File Systems Management**]: Access methods, D.4.6 [**Security and Protection**]: Information flow controls

General Terms

Design, Security

Keywords

Secure deletion, file systems, storage, security, NAND flash

1. INTRODUCTION

Data privacy is an increasing concern as more sensitive information is being stored in electronic devices. Once sensitive data is no longer needed, users may wish to permanently remove this data from electronic storage. However, typical file deletion mechanisms only update the file's metadata (e.g., pointers to the data), leaving the file data intact. Even recreating the file system from scratch does not ensure the data is removed [7].

A remedy is secure deletion, which should render data irrecoverable. Much secure deletion is implemented through partition- or device-wide encryption or overwriting techniques (see §2). However, coarse-grained methods may not work on media such as NAND flash [41], and are cumbersome to use when only a few files need to be securely deleted. Further, securely deleting an entire device or partition may be infeasible for embedded devices.

Per-file secure deletion is concerned with securely removing a specific file's content and metadata (e.g., name). This ability can assist with implementing privacy policies concerning the selective destruction of data after it has expired (e.g., client files), complying with government regulations to dispose of sensitive data (e.g., HIPPA [10]), deleting temporarily shared trade secrets, military applications demanding immediate destruction of selected data, and disposing of media in one-time-use applications.

Unfortunately, existing per-file secure-deletion solutions tend to be file-system- and storage-medium-specific, or limited to one segment of the operating-system storage data path (e.g., the file system) without taking into account other components (e.g., storage media type). For example, a secure deletion issued by a program might not be honored by optimization software used on typical flash devices that keep old versions of the data [41]. Solutions that rely on secure deletion of a stored encryption key become a subset of this problem, because they, too, must have a way to ensure the key is erased.

In addition, achieving secure deletion is hard due to diverse threat models. This paper focuses on dead forensics attacks on local storage, which occur after the computer has been shut down properly. Attacks on backups or live systems, cold-boot attacks [9], covert channels, and policy violations are beyond our scope.

In particular, our system assumes that we have full control of the entire storage data path in a non-distributed environment. Thus, the research question is, under benign user control and system environments, what holistic solution can we design and build to ensure that the secure deletion of a file is honored throughout the legacy storage data path? Although tightly constrained, this criterion still presents significant challenges.

We introduce TrueErase, a framework that irrevocably deletes data and metadata upon user request. TrueErase goes to the heart of the user's mental model: securely deleted data, like a shredded document, should be irrecoverable.

We assume the presence of file-system-consistency properties [34], which have been shown to be a requirement of secure deletion. We also note that if we have control over the storage layer, achieving raw NAND flash secure deletion is straightforward [18, 27, 38, 41].

However, TrueErase is designed to overcome the many challenges of correctly propagating secure deletion information in a full-data-path manner—all the way from the user to the storage. This framework is essential for building other secure-deletion capabilities that rely on tainting or encryption-based key deletion. Thus, our contributions are the following: (1) a per-file secure-deletion framework that works with the legacy storage data path, which (2) addresses the challenges raised throughout the data path into a single work and (3) has been systematically verified.

2. EXISTING APPROACHES

We distinguish the need for TrueErase from existing approaches by enumerating desirable characteristics of secure deletion systems (Table 1).

- ***Per-file:*** Fine-granularity secure deletion brings greater control to the user while potentially reducing costly secure deletion operations on the storage device.

ACSAC '12, Dec. 3–7, 2012, Orlando, Florida, USA.

- ***Encryption-free:*** Encryption systems securely delete by destroying the keys of encrypted files. However, these systems are complex due to managing per-file keys, providing efficient random access, data and metadata padding and alignments, etc. Encryption may also expire due to more computing power (e.g., quantum computers) or implementation bugs [4, 5]; thus, encrypted data might still be recovered after its key is deleted. In addition, passphrase-derived keys can be surrendered (e.g., via coercion) to retrieve deleted files. For these reasons, systems that keep keys on storage or third-party sites still need a way to securely delete keys on various storage media.
- ***Data-path-wide:*** Many secure-deletion solutions are within one layer of the legacy storage data path (Figure 1). The storage-management layer has no information about a block's file ownership [6, 35] and cannot support per-file deletion. File-system- and application-layer solutions are generally unaware of the storage medium and have limited control over the storage location of data and metadata. In addition, storage such as NAND flash may leave multiple versions of sensitive data behind. Thus, we need a solution that spans the entire data path to enforce secure deletion.
- ***Storage-medium-agnostic:*** A secure-deletion framework should be general enough to accommodate solid-state storage and portable devices that use these types of storage.
- ***Limited changes to the legacy code:*** Per-file secure deletion requires some information from the file system. However, getting such information should not involve modifying thousands of lines of legacy code.
- ***Metadata secure deletion***: Secure deletion should also remove metadata, such as the file name, size, etc.
- ***Crash handling:*** A secure-deletion solution should anticipate system crashes and provide meaningful semantics afterwards.

Table 1: Existing secure-deletion methods and characteristics. columns: F. per-file; E. encryption-free; D. data-path-wide; S. storage-medium-agnostic; L. limited changes to legacy code; M. securely delete metadata; C. handle crashes

	F	E	D	S	L	M	C
Secure delete encrypted device/partition key [12, 13, 19, 30, 40]					✓	✓	✓
Specialized hard drive commands [11]		✓			✓	✓	✓
Specialized flash medium commands (page granularity) [41]	✓	✓			✓		
Stackable file system deletion [14, 15]	✓	✓			✓	✓	✓
Modified file system – deletion through overwriting [1, 14, 15]	✓	✓			✓	✓	✓
Modified file system – deletion through encryption [25]	✓				?		?
Dedicated server(s) for encryption keys [8, 24]	✓				✓	?	✓
Encrypted backup system [2]	✓				✓		?
User-space solution on top of flash file system [27]	✓	✓			✓	?	
Overwriting tools [19, 28, 29, 31, 42]	✓	✓			✓		
Modified flash file systems – device erasures and/or overwriting [27, 38]	✓	✓	✓		✓	?	?
Modified flash file systems – encryption with key erasure [18]	✓		✓		?	✓	?
Semantically-Smart Disk Systems [35, 36]	✓	✓	✓		✓		✓
Type-Safe Disks [33]	✓	✓	✓				✓
Data Node Encrypted File System [26]	✓	✓	✓		✓	✓	✓
TrueErase	✓	✓	✓	✓	✓	✓	✓

3. CHALLENGES AND ASSUMPTIONS

Designing and implementing per-file secure deletion is challenging for a number of reasons:

- ***No pre-existing deletion operation:*** Other than removing references to data blocks and setting the file size and allocation bits to zeros, file systems typically do not issue requests to erase file content. (Note that the ATA Trim command was not implemented for security [32] and might not delete all data [17].)
- ***Complex storage-data-path optimizations***: Secure deletion needs to retrofit legacy asynchronous optimizations. In particular, storage requests may be reordered, concatenated, split, consolidated (applying one update instead of many to the same location), cancelled, or buffered while in transit.
- ***Lack of data-path-wide identification***: Tracking sensitive data throughout the operating system is complicated by the possible reuse of data structures, ID numbers, and memory addresses.
- ***Verification***: Although verification is often overlooked by various solutions, we need to ensure that (1) secure deletions are correctly propagated throughout the storage data path, and (2) assumptions are checked when possible.

Due to these challenges, we assume a user has administrative control of an uncompromised, single-user, single-file-system, non-RAID, non-distributed system. The threat model is dead forensics attacks, which occur after a user unmounts and shuts down the system after completing secure-deletion operations.

Our system assumes control of the entire storage data path. Although assuming access to proprietary firmware may be optimistic, this paper argues that there is a need for storage devices to expose information and control to support secure-deletion features correctly.

In addition, we assume that common journaling file systems that adhere to the consistency properties specified by [34] are used, since we cannot verify secure deletion in file systems that cannot even guarantee their own consistency (e.g., ext2, FAT). Further, all update events and block types are reported to our framework to verify that we are tracking all important events. These assumptions allow us to focus on building and verifying the properties of secure deletion.

4. TRUEERASE DESIGN

We introduce TrueErase (Figure 1), a holistic framework that propagates file-level information to the block-level storage-management layer via an auxiliary communication path, so that per-file secure deletion can be honored throughout the data path.

Our system is designed with the following observations and formulated guidelines:

- Modifying the request ordering of the legacy data path is undesirable because it is difficult to verify the legacy semantics. Thus, we leave the legacy data flow intact.
- Per-file secure deletion is a performance optimization over applying secure deletion to all file removals. Thus, we can simplify tricky decisions about whether a case should be handled securely by handling it securely by default.
- Persistent states complicate system design with mechanisms to survive reboots and crashes. Thus, our solution minimizes the use of persistent states.

The major areas of TrueErase's design include (1) a user model to specify which files or directories are to be securely deleted, (2) tracking and propagating information across storage layers via a centralized module named ***TAP***, (3) enforcing secure deletion via storage-specific mechanisms added to the storage-management layer, and (4) exploiting file-system consistency properties to enumerate cases for verification.

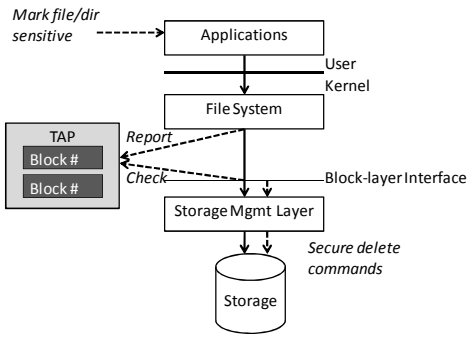

Figure 1. TrueErase framework. The shaded box and dotted lines represent our augmented data path. Other boxes and lines represent the legacy storage data path.

4.1 User Model

A naïve user may mark the entire storage device sensitive for secure deletion. A more sophisticated user can mark all user-modifiable folders sensitive. An expert user can follow traditional permission semantics and apply common attribute-setting tools to mark a file or directory as ***sensitive***, which means the sensitive file, or all files under the sensitive directory, will be securely deleted when removed. A legacy application can then issue normal deletion operations, which are carried out securely, to remove sensitive file and directory data content, metadata, and associated copies within the storage data path. However, there are deviations from the traditional permission models.

Toggling the sensitive status: Before the status of a file or directory is toggled from non-sensitive to sensitive, older versions of the data and metadata may have already been left behind. Without tracking all file versions or removing all old versions for all files, TrueErase can enforce secure deletion only for files or directories that have remained sensitive since their creation. Should a non-sensitive file be marked sensitive, secure deletion will be carried out only for the versions of the metadata and file content created after that point.

Name handling: A directory is traditionally represented as a special file, with its data content storing the file names the directory holds. Although file permissions are applied to its data content, permission to handle the file name is controlled by its parent directory.

Under TrueErase, marking a file sensitive will also cause its name stored in the parent directory to be securely handled, even if the parent is not marked sensitive. Otherwise, marking a file sensitive would require its parent to be marked sensitive for containing the sensitive file name, its grandparent to be marked sensitive for containing the sensitive parent name, etc., until reaching the root directory.

Links: Similar to legacy semantics, a secure deletion of a hard link is performed when the link count decreases to zero. Symbolic link names and associated metadata are not supported, but file data written to a symbolic link will be treated sensitively if the link's target file is sensitive.

4.2 Information Tracking and Propagation

Tracking and propagating information from the file system to the lower layers is done through a centralized type/attribute propagation (***TAP***) module.

4.2.1 Data Structures and Globally Unique IDs

TAP expands the interface between the file system and the block layer in a backward-compatible way. This passive forwarding module receives pending sensitive update and deletion events from file systems, and uses internal ***write entries*** and ***deletion reminders,*** respectively, to track these events.

Since various data structures (e.g., block I/O structures), namespaces (e.g., i-node numbers), and memory addresses can be reused, correct pairing of write entries and deletion reminders requires a unique ID scheme. In particular, the IDs need to be (1) accessible throughout the data path and (2) altered when its content association changes. TrueErase embeds a monotonically increasing globally unique page ID in each memory page structure accessible throughout the data path. The ID is altered at allocation time, so that the same page reallocated to hold versions of the same logical block has different IDs.

To handle various tracking units, such as logical blocks, requests, physical sectors, and device-specific units, TAP tracks by the physical sector, because that is unique to the storage device and can be computed from anywhere in the data path. The globally unique page ID and the physical sector number form a globally unique ID (***GUID***).

In addition to the GUID, a write entry contains the update type (e.g., i-node) and the security status. A deletion reminder contains a ***deletion list*** of physical sector numbers to be deleted.

Table 2: The TAP Interface.

TE_report_write(*GUID, block type, secure status*): Creates or updates a TAP write entry associated with the GUID.
TE_report_delete(*metadata GUID, metadata block type, metadata secure status, deletion sector number*): Creates or updates a TAP deletion reminder that contains the sector number. The reminder is attached to the write entry (created as needed) associated with the metadata GUID.
TE_report_copy(*source GUID, destination GUID, flag*): Copies information from a write entry corresponding to the source GUID to a new entry corresponding to the destination GUID. The flag determines whether deletion reminders should be transferred.
TE_cleanup_write(*GUID*): Removes the write entry with a specified GUID. This call also handles the case in which a file has already been created, written, and deleted before being flushed to storage.
TE_check_info(*GUID*): Used to query TAP to retrieve information about a block layer write with a specific GUID.

4.2.2 TAP Interface and Event Reporting

Through the TAP interface (Table 2), the file system must report all important events: file deletions and truncations, file updates, and certain journaling activities.

File deletions and truncations: Data deletion depends on the update of certain metadata, such as free-block bitmaps. TAP allows file systems to associate deletion events with metadata update events via TE_report_delete(). Within TAP, write entries are associated with respective deletion reminders and their deletion lists.

Eventually, the file system flushes the metadata update. Once the block-layer interface receives a sector to write, the interface uses the GUID to look up information in TAP through TE_check_info(). If the write entry is linked to a deletion reminder, the storage-management layer must securely delete those sectors on the deletion list before writing the metadata update.

Additionally, the storage-management layer can choose a secure-deletion method that matches the underlying medium. For NAND flash, triggering the erase command once may be sufficient (details in §4.3). For a hard drive, the sectors can be directly overwritten with random data.

After secure deletion has been performed and the metadata has been written to storage, the block-layer interface can call TE_cleanup_write() to remove the associated TAP write entries and deletion reminders.

For data-like metadata blocks (e.g., directory content), deletion handling is the same as that of data blocks.

File updates: Performing deletion operations is not enough to ensure all sensitive information is securely deleted. By the time a secure-deletion operation is issued for a file, several versions of its blocks might have been created and stored (e.g., due to flash optimizations), and the current metadata might not reference old versions. One approach is to track all versions, so that they can be deleted at secure-deletion time. However, tracking these versions requires persistent states, and thus recovery mechanisms to allow those states to survive failures.

TrueErase avoids persistent states by tracking and deleting old versions along the way. That is, secure deletion is applied for each update that intends to overwrite a sensitive block in place (***secure write*** for short). Therefore, in addition to deletion operations, TrueErase needs to track all in-transit updates of sensitive blocks.

The file system can report pending writes to TAP through TE_report_write(). Eventually, the block-layer interface will receive a sector to write and can then look up information via TE_check_info(). If the corresponding write entry states that the sector is secure, the storage-management layer will write it securely. Otherwise, it will be written normally. After the sector is written, the block-layer interface calls TE_cleanup_write().

4.2.3 Journaling and Other Events

Sometimes file blocks are copied to other memory locations for performance and accessibility reasons (e.g., file-system journal, bounce buffers, etc.). When that happens, any write entries associated with the original memory location must be copied and associated with the new location. If there are associated deletion reminders, whether they are transferred to the new copy is file-system-specific. For example, in ext3, the deletion reminders are transferred to the memory copy in the case of bounce buffers or to the memory copy that will be written first in the case of journaling. Memory copies are reported to TAP through TE_report_copy().

4.3 Enhanced Storage-management Layer

TrueErase does not choose a secure-deletion mechanism until a storage request has reached the storage-management layer. By doing so, TrueErase ensures that the chosen mechanism matches the characteristics of the underlying storage medium. For example, the process of securely erasing flash memory (erase) is quite different from the process of securely erasing information on a disk (overwrite). Users should be unaware of this difference.

In addition, we can further add different secure-deletion methods at the storage layer in accordance with different requirements and government regulations [21, 37].

TrueErase can work with both flash storage and traditional hard drives. Due to the difficulty of secure deletion on flash, we concentrate on applying TrueErase to flash storage in this paper. However, we also provide a high-level design of a hard drive solution to show the generalizability of TrueErase.

4.3.1 NAND Flash Storage Management

NAND flash basics: NAND flash has the following characteristics: (1) writing is slower than reading, and erasure can be more than an order of magnitude slower [3]; (2) NAND reads and writes are in ***flash pages*** (of 2-8 Kbytes), but erasures are performed in ***flash blocks*** (typically 64-512 Kbytes consisting of contiguous pages); (3) in-place updates are generally not allowed—once a page is written, the flash block containing this page must be erased before this page can be written again, and other in-use pages in the same flash block need to be copied elsewhere; and (4) each storage location can be erased only 10K-1M times [3].

As a common optimization, when flash receives a request to overwrite a flash page, a ***flash translation layer (FTL)*** remaps the write to a pre-erased flash page, stamps it with a version number, and marks the old page as invalid, to be cleaned later. (Flash overwrites might be allowed for some special cases.) These invalid pages are not accessible to components above the block layer; however, they can be recovered by forensic techniques [20]. To prolong the lifespan of the flash, ***wear-leveling*** techniques are often used to spread the number of erasures evenly across all storage locations.

NAND secure commands: We added two secure-deletion commands to the storage-management layer (i.e., FTL) for flash.

Secure_delete(*page numbers*): This call specifies pages on flash to be deleted securely. The call copies other in-use pages from the current flash block to other areas, and marks those pages as unused in the block. The specified pages then can be marked invalid, and the current flash block can be cleared via a flash erase command.

Secure_write(*page numbers, data*): Generally, writing to the same logical page to flash would result in a new physical page being allocated and written, with the physical page holding old data versions marked invalid. This call, on the other hand, would securely delete those invalid pages with Secure_delete().

We choose this type of secure-deleting flash behavior instead of zero-overwriting or scrubbing [27, 38, 41] for ease of implementation and portability. Alternative flash secure deletion schemes may have better performance on specific chips.

NAND garbage collection: When a NAND flash device runs low on space, it triggers wear leveling to compact in-use pages into fewer flash blocks. However, this internal storage reorganization does not consult with higher layers and has no knowledge of file boundaries, sensitive status, etc. Thus, in addition to storing a file's sensitive status in the extended attributes, we found it necessary to store a sensitive-status bit in the per-page ***control area***. (This area also contains a page's in-use status.) With this bit, when a sensitive page is migrated to a different block, the old block is erased via Secure_delete(). Consistency between the file system and the storage status is addressed in § 4.5.

4.3.2 Hard Drive Storage Management

Hard drive basics: On a hard drive, writes can be performed in-place, in that a write to a logical sector will directly overwrite the same physical sector.

Hard drive secure commands: Similarly to the NAND secure commands, we can add **secure_delete**(*sector numbers*) and **secure_write**(*sector numbers, data*) commands for disk. The results of these commands are the same as for the flash case—only the mechanism changes. The **secure_delete** command will overwrite *sector numbers* in-place with random data, and the **secure_write** command will first overwrite sectors specified by *sector numbers* with random data before writing *data.* The number of overwrites of random data is configurable. These commands can be placed in a high-level software driver, such as the Linux device-mapper layer.

4.4 File-system-consistency Properties

We cannot guarantee the correctness of our framework if it is required to interact with file systems that exhibit arbitrary behaviors. Therefore, we applied the consistency properties of file systems defined in [34] to enumerate corner cases. (These

properties also rule out ext2 and FAT, which are prone to inconsistencies due to crashes.) By working with file systems that adhere to these properties, we can simplify corner-case handling and verify our framework systematically.

In a simplified sense, as long as pieces of file metadata reference the correct data and metadata versions throughout the data path, the system is considered consistent. In particular, we are interested in three properties in [34]. The first two are for non-journaling-based file systems. In a file system that does not maintain both properties, a non-sensitive file may end up with data blocks from a sensitive file after a crash recovery. The last property is needed only for journaling-based file systems.

- ***The reuse-ordering property*** ensures that once a file's block is freed, the block will not be reused by another file before its free status becomes persistent. Otherwise, a crash may lead to a file's metadata pointing to the wrong file's content. Thus, before the free status of a block becomes persistent, the block will not be reused by another file or changed into a different block type. With this property, we do not need to worry about the possibility of dynamic file ownership and types for in-transit blocks.
- ***The pointer-ordering property*** ensures that a referenced data block in memory will become persistent before the metadata block in memory that references the data block. With this ordering reversed, a system crash could cause the persistent metadata block to point to a persistent data block location not yet written. This property does not specify the fate of updated data blocks in memory once references to the blocks are removed. However, the legacy memory page cache prohibits unreferenced data blocks from being written. The pointer-ordering property further indicates that right after a newly allocated sensitive data block becomes persistent, a crash at this point may result in the block being unreferenced by its file. To cover this case, we will perform secure deletion on unreferenced sensitive blocks at recovery time (see §4.5).
- ***The non-rollback property*** ensures that older data or metadata versions will not overwrite newer versions persistently—critical for journaling file systems with versions of requests in transit. That is, we do not need to worry that an update to a block and its subsequent secure deletion will be written persistently in the wrong order.

With these consistency properties, we can identify the structure of secure-deletion cases and handle them by: (1) ensuring that a secure deletion occurs before a block is persistently declared free, (2) the dual case of hunting down the persistent sensitive blocks left behind after a crash but before they are persistently referenced by file-system metadata, (3) making sure that secure deletion is not applied (in a sense, too late) to the wrong file, (4) the dual case of making sure that a secure block deletion is not performed too early and gets overwritten by a buffer update from a deleted file, and (5) handling in-transit versions of a storage request (mode changing, reordering, consolidation, merging, and splitting). Buffering, asynchrony, and cancelling of requests are handled by TAP.

4.5 Other Design Points

Crash handling: TAP contains no persistent states and requires no additional recovery mechanisms. Persistent states stored as file-system attributes are protected by journal-recovery mechanisms. At recovery time, a journal is replayed with *all* operations handled securely. We then securely delete the entire journal. To hunt down leftover sensitive data blocks, we sequentially delete sensitive blocks that are not marked as allocated by the file system for flash and disk [1]. Since flash migrates in-use pages from to-be-erased blocks by copying those pages elsewhere before erasing the old versions, some sensitive pages may have duplicates during a crash. Given that the secure deletion of the page did not complete, the common journal crash-recovery mechanism will reissue the operation, so that the other remaining in-use pages in the same flash block can continue the migration, and the block can then be erased.

***Consolidation of requests*:** When consolidation is not permitted (e.g., consolidations of overwrites on storage), we need to disable storage-built-in write caches or use barriers and device-specific flush calls to ensure that persistent updates are achieved [22]. When consolidation is permitted (such as in the page cache or journal), we interpret an update's sensitive status during consolidation conservatively. As long as one of the updates to a given location is sensitive, the resulting update will be sensitive.

Dynamic sensitive-mode changes for in-transit blocks: To simplify tracking the handling of a block's sensitive status, we allow a non-sensitive in-transit file or directory to be marked sensitive, but a sensitive object is not allowed to be marked non-sensitive.

Shared block security status: A metadata block often stores metadata for many files, probably with mixed sensitive status. Thus, updating non-sensitive metadata may also cause the sensitive metadata stored on the same block to appear in the data path. We simplify the handling of this case by treating a shared metadata block sensitively as long as one of its metadata entries is sensitive.

Partial secure deletion of a metadata sector: Securely deleting a file's metadata only at the storage level is insufficient, since a metadata block shared by many files may still linger in the memory containing the sensitive metadata. If the block is written again, the metadata we thought securely deleted will return to storage. In these cases, we zero out the sensitive metadata within the block in memory, and then update the metadata as a sensitive write.

5. TRUEERASE IMPLEMENTATION

We prototyped TrueErase under Linux 2.6.25.6 and applied our framework to the popular ext3 and jbd journaling layer due to their adherence to file-system-consistency properties [34]. Because raw flash devices and their development environments were not widely available when we began our research, we used SanDisk's DiskOnChip flash and the associated Inverse NAND File Translation Layer (INFTL) kernel module as our FTL. Although DiskOnChip is dated, our design is applicable to modern flash and development environments. As future work, we will explore newer environments, such as OpenSSD [39].

Overall, our user model required 198 lines of C code; TAP, 939; secure-deletion commands for flash, 592; a user-level development environment for kernel code, 1,831; and a verification framework, 8,578.

5.1 Secure-deletion Attributes

A user can use the legacy `chattr +s` command to mark a file or directory as sensitive. However, by the time a user can set attributes on a file, its name may already be stored non-sensitively. Without modifying the OS, one remedy is to cause files or directories under a sensitive directory to inherit the attribute when they are created. We also provide `smkdir` and `screat` wrapper scripts that create a file or directory with a temporary name, mark it sensitive, and rename it to the sensitive name.

5.2 TAP Module

TAP is implemented as a kernel module. We will give a brief background on ext3 and jbd to clarify their interactions with TAP.

5.2.1 Background on Ext3, Journaling, and Jbd

File truncation/deletion under ext3: Ext3 deletes the data content of a file via its truncate function, which involves updating ***(t1)*** the i-node to set the file size to zero, ***(t2)*** metadata blocks to remove pointers to data blocks, and ***(t3)*** the bitmap allocation blocks to free up blocks for reuse. Multiple rounds of truncates may be required to delete the content of a large file.

Deleting a file involves: removing the name and i-node reference from the directory, adding the removed i-node to an orphan list; truncating the entire file via (t1) to (t3); removing the i-node from the orphan list; and updating the i-node map to free the i-node.

Journaling: Typical journaling employs the notion of a transaction, so the effect of an entire group of updates is either all or nothing. With group-commit semantics, the exact ordering of updates within a transaction (an update to an i-node allocation bitmap block) may be relaxed while preserving correctness, even in the face of crashes. To achieve this effect, all writes within a transaction are ***(j1)*** journaled or committed to storage persistently; then ***(j2)*** propagated to their final storage destinations, after which ***(j3)*** they can be discarded from the journal.

A committed transaction is considered permanent even before its propagation. Thus, once a block is committed to be free (j1), it can be used by another file. At recovery time, committed transactions in the journal are replayed to re-propagate or continue propagating the changes to their final destinations. Uncommitted transactions are aborted.

Jbd: Jbd differentiates data and metadata. We chose the popular ordered mode, which journals only metadata but requires ***(j0)*** data blocks to be propagated to their final destination before the corresponding metadata blocks are committed to the journal.

5.2.2 Deployment Model

All truncation, file deletion, and journaling operations can be expressed and performed as secure writes and deletions to data and metadata blocks. The resulting deployment model and its applicability are similar to those of a journaling layer. We inserted around 60 TAP-reporting calls in ext3 and jbd, with most collocated with block-layer interface write submission functions and various dirty functions (e.g., ext3_journal_dirty_data).

Applicable block types: Secure writes and deletions are performed for sensitive data, i-node, extended-attribute, indirect, and directory blocks, and corresponding structures written to the journal. Remaining metadata blocks (e.g., superblocks) are frequently updated and shared among files (e.g., bitmaps) and do not contain significant information about files. By not treating these blocks sensitively, we reduce the number of secure-deletion operations.

Secure data updates (Figure 2): Ext3/jbd calls TE_report_write() on sensitive data block updates, and TAP creates per-sector write entries. Updates to the same TAP write entries are consolidated via GUIDs; this behavior reflects that of the page cache.

The data update eventually reaches the block-layer interface (via commit), which retrieves the sensitive status via TE_check_info(). The layer can then perform the secure-write operation and invoke TE_cleanup_write() to remove the corresponding write entries.

Secure metadata updates: A metadata block must be securely written to and deleted from the journal. Ext3 reports pending journal writes to TAP via TE_report_copy().

Jbd manages its in-use persistent journal locations through its own superblock allocation pointers and a clean-up function, which can identify locations no longer in use. Through TE_report_delete(), we can put those locations on the deletion list and associate them with the journal superblock update. After the journal superblock is securely updated, the locations on the deletion list can be securely wiped. In the case of a crash, we securely delete all journal locations through TE_report_delete() once all committed updates have been securely applied.

Secure data deletions (Figure 3): When deleting sensitive file content, ext3's truncate function informs TAP of the deletion list and associated file i-node via TE_report_delete(). Given the transactional semantics of a journal (§5.2.1), we can associate the content-deletion event with the file's i-node update event instead of the free-block bitmap update event. Thus, we securely delete data before step ***(t1)***.

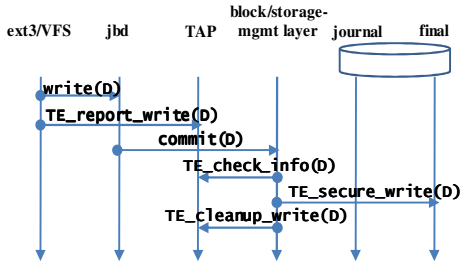

Figure 2. Secure data updates. D is the data block in various stages of being securely written.

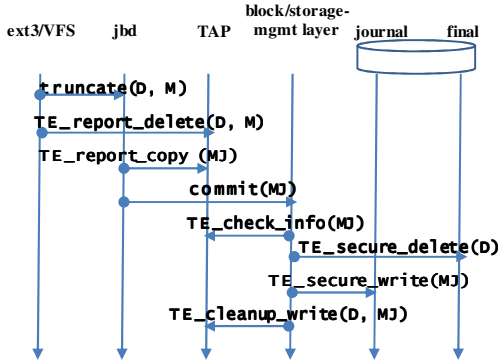

Figure 3. Secure data deletion. M is the updated metadata block; D is the data block in various stages of secure deletion; MJ is the metadata journal block that corresponds to the updated metadata block M.

TAP will create the i-node write entry and pair it with the corresponding secure-deletion reminder to hold the deletion list. When the write entry is copied via TE_report_copy(), reminders are transferred to the journal copy to ensure that secure deletions are applied to the matching instance of the i-node update.

When the block-layer interface receives the request to commit the update of the sensitive i-node to the journal, the interface calls TE_check_info() and retrieves the sensitive status of the i-node, along with the deletion list. The data areas are then securely deleted before the i-node update is securely written to the journal.

Secure metadata deletions: During a file truncation or deletion, ext3 also deallocates extended attribute block(s) and indirect block(s). Those blocks are attached to the i-node's list of secure-deletion reminders as well.

To securely delete an i-node or a file name in a directory, the block containing the entry is securely updated and reported via TE_report_write(). Additionally, we need to zero out the i-node and variable-length file name in the in-memory copies, so that they will not negate the secure write performed at the storage-management layer.

If a directory is deleted, its content blocks will be deleted in the same way as the content from a file.

Miscellaneous cases: Committed transactions might not be propagated instantly to their final locations. Across committed transactions, the same metadata entry (e.g., i-node) might have changed file ownership and sensitive status. Thus, jbd may consolidate, say, a non-sensitive update 1, sensitive update 2, and non-sensitive update 3 to the same location into a non-sensitive update. As a remedy, once a write entry is marked sensitive, it remains sensitive until securely written.

5.3 Enhanced FTL Storage-management Layer

We modified the existing Linux INFTL to incorporate secure deletion. INFTL uses a stack-based algorithm to remap logical pages to physical ones.

5.3.1 INFTL Extensions and Optimizations

INFTL remaps at the flash block level, where each 16-Kbyte flash block contains 32 512-byte pages, with a 16-byte control area per page. A remapped page always has the same offset within a block.

A NAND page can be in three states: empty, valid with data, or invalid. An empty page can be written, but an invalid page has to be erased to become an empty page.

***INFTL in-place updates*:** INFTL uses a stack of flash blocks to provide the illusion of in-place updates. When a page P1 is first written, an empty flash block B1 is allocated to hold P1. If P1 is written again (P1'), another empty flash block B2 is allocated and stacked on top of B1, with the same page offset holding P1'. Suppose we write P2, which is mapped to the same block. P2 will be stored in B2 because it is at the top of the stack, and its page at page offset for P2 is empty.

The stack will grow until the device becomes full; it will then be flattened into one block containing only the latest pages to free up space for garbage collection.

***INFTL reads*:** For a read, INFTL traverses down the appropriate stack from the top and returns the first valid page. If the first valid page is marked deleted, or if no data are found, INFTL will return a page of zeros.

Secure-deletion extensions: Our added secure write command is similar to the current INFTL in-place update. However, if a stack contains a sensitive page, we set its maximum depth to 1 (0 is the stack top). Once it reaches the maximum, the stack must be consolidated to depth 0. When consolidating, old blocks are immediately erased via the flash erase command, instead of being left behind.

Since the existing stack algorithm already tracks old versions, we also implemented the delayed-deletion optimization, which allows data blocks to defer the secure-write consolidation to file deletion time. Basically, the maximum depth is no longer bounded. Delaying secure deletion for metadata is trickier and will be investigated in future work.

A secure delete is a special case of a secure write. When a page is to be securely deleted, an empty flash block is allocated on top of the stack. All the valid pages, minus the page to be securely erased, are copied to the new block. The old block is then erased.

5.3.2 Disabled storage-management optimizations

Because jbd does not allow reordering to violate file system constraints and our flash has no built-in cache, we do not disable these optimizations.

6. VERIFICATION

We (1) tested the basic cases, assumptions, and corner cases discussed in §4.4 and (2) verified the state space of TAP.

6.1 Basic Cases

Sanity checks: We verified common cases of secure writes and deletes for empty, small, and large files and directories using random file names and sector-aligned content. After deletion, we scanned the raw storage and found no remnants of the sensitive information. We also traced common behaviors involving sensitive and non-sensitive objects; when the operation included a source and a destination, we tested all four possible combinations. The operations checked included moving objects to new directories, replacing objects, and making and updating symbolic and hard links. We also tested sparse files. In all cases, we verified that the operations behaved as expected.

Simulation of workload: We ran the PostMark benchmark [16] with default settings, modified with 20% of the files marked sensitive, with random content. Afterwards, we found no remnants of sensitive information.

Missing updates: To check that all update events and block types are reported, we looked for errors such as unanticipated block-type changes and unfound write entries in TAP, etc., which are signs of missing reports from the file system. Currently, all updates are reported.

Cases related to file-system-consistency properties: For cases derived from the reuse-ordering property, we created an ext3 file system with most of its i-nodes and blocks allocated, to encourage reuse. Then we performed tight append/truncate and file creation/deletion loops with alternating sensitive status. We used uniquely identifiable file content to detect sensitive information leaks and found none.

For pointer-ordering-related cases, we verified our ability to recover from basic failures and remove remnants of sensitive information. We also verified that the page cache prohibits unreferenced data blocks from being written to the storage.

Since the page ID part of GUIDs increases monotonically, we can use this property to detect illegal reordering of sensitive updates for the cases derived from the non-rollback property. For consolidations within a transaction, we used tight update loops with alternating sensitive modes. For consolidations across transactions, we used tight file creation/deletion loops with alternating sensitive modes. We checked all consolidation orderings for up to three requests (e.g., non-sensitive/sensitive/non-sensitive).

6.2 TAP Verification

We enumerated the TAP state-transition table and verified its correctness via two-version programming.

State representation: We exploited TAP's properties to trim the state space. First, a write entry will not consolidate with other write entries. This property ensures that each sensitive update is carried out unless explicitly cancelled. Various consolidation behaviors (e.g., page cache) are achieved by performing updates directly to the write entry. Second, the next state transition is based on current write entries of different types within a current state (plus inputs). With those two properties, we can reduce the representation of a state to at most one write entry of each type, and explore all state-generating rules.

To illustrate, each state holds one write entry for nine block types: data, i-node, other metadata, journal copy of data, journal copy of i-node, journal copy of other metadata, copy of data, copy of i-node, and copy of other metadata. Additionally, each write entry has four status bits: allocated, sensitive, having reminder attached, and ready to be deleted from the journal. Thus, a state is a 9x4 matrix and can be represented as 36 bits, with 2^{36} states.

State transitions: Each interface call triggers a state transition based on the input parameters. For example, the first TE_report_write() on a non-sensitive i-node will transition from the empty state (a zero matrix), say S0, to a state S1, where the allocated bit for the i-node is set to 1. If TE_report_write() is called again to mark the i-node as sensitive, S1 is transitioned to a new state S2, with allocated and sensitive bits set to 1s.

State-space enumeration: To enumerate states and transitions, we permuted all TAP interface calls with all possible input parameters to the same set of write entries. A small range of GUIDs was used so that each write entry could have a unique GUID, but GUID collisions were allowed to test error conditions. Given that the enumeration step can be viewed as traversing a state-space tree in breadth-first order, the tree fanout at each level is the total number of interface call-parameter combinations (261). As an optimization, we visited only states reachable from the starting empty state, and avoided repeated state-space and sub-tree branches. As a result, we explored a tree depth of 16 and located ~10K unique reachable states, or ~2.7M state transitions.

Two-version-programming verification: Given that the state-transition table is filled with mostly illegal transitions, we applied n-version programming to verify the table, where the probability of hitting the same bug with the same handling can be reduced as we add more versions. In this work, n=2. We wrote a user-level state-transition program based on hundreds of conceptual rules (e.g., marking a write entry of any type as sensitive will set the sensitive bit to 1). The enumerated state-transition table was reconciled with the one generated by the TAP kernel module.

7. PERFORMANCE EVALUATION

We compared TrueErase to an unmodified Linux 2.6.25.6 running ext3. We ran PostMark [16] to measure the overhead for metadata-intensive small-file I/Os. We also compiled OpenSSH version 5.1p1 [23] to measure the overhead for larger files. We ran our experiments on an Intel® Pentium® D CPU 2.80GHz dual-core Dell OptiPlex GX520 with 4-GB DDR533 and 1-GB DoC MD2203-D1024-V3-X 32-pin DIP mounted on a PCI-G DoC evaluation board. Each experiment was repeated 5 times. The 90% confidence intervals are within 22%.

PostMark: We used the default configuration with the following changes: 10K files, 10K transactions, 1-KB block size for reads and writes, and a read bias of 80%. We also modified PostMark to create and mark different percentages of files as sensitive. These files can be chosen randomly or with spatial locality, which is approximated by choosing the first x% of file numbers. Before running tests for each experimental setting, we dirtied our flash by running PostMark with 0% sensitive files enough times to trigger wear leveling. Thus, our experiments reflect a flash device operating at steady state. A `sync` command was issued after each run and is reflected in the elapsed time.

Table 3 shows that when TrueErase operates with no sensitive files, metadata tracking and queries account for 3% overhead compared to the base case. With 10% of files marked sensitive, the slowdown factor can be as high as 11, which confirmed the numbers in a prior study [41]. However, with 5% of files marked sensitive and with locality and delayed secure deletion of file data blocks, the slowdown factor can be reduced to 3.4, which is comparable to disk-based secure-deletion numbers [14].

We noticed some feedback amplification effects. Longer runs mean additional memory page flushes, which translate into more writes, which involve more reads and erases as well and lead to even longer running times. Thus, minor optimizations can improve performance significantly.

Table 3: Postmark flash operations, times, and overhead percentage compared to base.

	page reads	control-area reads	page writes	control-area writes	erases	time (secs)
Base	300K	1.97M	218K	237K	4.28K	671
0%	0.99x	1.08x	1.01x	1.01x	1.00x	**1.03x**
1% random	3.69x	2.09x	2.82x	2.79x	2.58x	2.93x
1% locality	2.95x	1.89x	2.33x	2.31x	2.16x	2.44x
1% random, delayed deletion	3.41x	2.00x	2.61x	2.59x	2.47x	2.73x
1% locality, delayed deletion	2.77x	1.77x	2.20x	2.19x	2.08x	2.29x
5% random	10.3x	4.22x	6.91x	6.83x	6.67x	7.39x
5% locality	6.69x	3.19x	4.86x	4.81x	4.32x	5.05x
5% random, delayed deletion	7.56x	3.48x	4.99x	4.99x	5.18x	5.54x
5% locality, delayed deletion	4.40x	2.33x	3.29x	3.29x	3.02x	**3.42x**
10% random	15.3x	5.82x	9.96x	9.84x	9.75x	**10.7x**
10% locality	9.96x	4.24x	7.00x	6.92x	6.23x	7.27x
10% random, delayed deletion	9.44x	4.23x	5.91x	5.96x	6.54x	6.80x
10% locality, delayed deletion	5.82x	2.96x	4.19x	4.22x	3.90x	4.45x

Table 4: Compilation flash operations, times, and overhead percentage compared to base.

	page reads	control-area reads	page writes	control-area writes	erases	time (secs)
make + sync						
Base	25.3K	108K	22.5K	23.9K	352	89
Random	4.79x	3.10x	3.15x	3.15x	3.13x	2.51x
Random, delayed deletion	1.70x	1.37x	1.41x	1.43x	1.40x	**1.41x**
make clean + sync						
Base	1.60K	3.73K	445	514	22	3
Random	10.0x	10.1x	13.6x	15.0x	7.14x	8.13x
Random, delayed deletion	8.47x	8.36x	11.0x	12.6x	6.22x	**6.87x**
Total						
Base	26.9K	112K	23.0K	24.4K	374	92
Random	5.10x	3.33x	3.35x	3.40x	3.37x	2.70x
Random, delayed deletion	2.10x	1.60x	1.59x	1.66x	1.69x	**1.59x**

OpenSSH compilations: We issued `make+sync` and `make clean+sync` to measure the elapsed times for compiling and cleaning OpenSSH [23]. For the TrueErase case, we marked the `openbsd-compat` directory sensitive before issuing `make`, which would cause all newly created files (e.g., .o files) in that directory to be treated sensitively. These files account for roughly 27% of the newly generated files (8.2% of the total number of files and 4.1% of the total number of bytes after compilation). Before running each set of tests, we dirtied the flash in the same manner as with PostMark and discarded the first run that warms up the page cache. Table 4 shows that a user would experience a compilation slowdown within 1.4x under the delay-deletion mode. A user would experience a slowdown within 6.9x under the delayed-deletion mode with a deletion-intensive workload. For the entire compilation cycle of `make+sync` with `make clean+sync`, a user would experience an overall slowdown within 60% under the delayed-deletion mode.

Overall, we find that the overhead is within our expectations. Further improvements in performance are future work.

8. RELATED WORK

This section discusses existing cross-layer secure-deletion solutions.

A semantically-smart-disk system (SDS) [36] observes disk requests and deduces common file-system-level information such as block types. The File-Aware Data-Erasing Disk is an ext2-based SDS that overwrites deleted files at the file-system layer.

A type-safe disk [33] directly expands the block-layer interface and the storage-management layer to perform free-space management. Using a type-safe disk, a modified file system can specify the allocation of blocks and their pointer relationships. As an example, this work implements secure deletion on ext2. Basically, when the last pointer to a block is removed, the block can be securely deleted before it is reused.

Lee et al. [18] have modified YAFFS, a log-structured file system for NAND flash, to handle secure file deletion. The modified YAFFS encrypts files and stores each file's key along with its metadata. Whenever a file is deleted, its key is erased, and the encrypted data blocks remain. Sun et al. [38] modified YAFFS and exploited certain types of NAND flash that allow overwriting of pages to achieve secure deletion. Raerdon et al. [27] also modified YAFFS to use a flash-chip-specific zero-overwriting technique. In addition, Raerdon et al. [26] developed the Data Node Encrypted File System (DNEFS), which modifies the flash file system UBIFS to perform secure deletion at the data node level, which is the smallest unit of reading/writing. DNEFS performs encryption and decryption of individual data nodes and relies on a key generation and deletion scheme to prevent access to overwritten or deleted data. Since UBIFS is designed for flash with scaling constraints, this approach is not as applicable for disks and larger-scale storage settings.

9. FUTURE WORK

Many opportunities exist to increase TrueErase's performance on NAND flash. We can implement flash-chip-specific zero-overwriting or scrubbing routines [27, 38, 41]. However, this optimization may make our solution less portable. We can add encryption to our system and use TrueErase to ensure secure deletion of the encryption key. We could also batch flash erasures for better flash performance.

Other future work will include tracking sensitive data between files and applications via tainting mechanisms, expanded handling of other threat models, and generalizations to handle swapping, hibernation, RAID, and volume managers.

10. LESSONS LEARNED/CONCLUSION

This paper presents our third version of TrueErase. Overall, we found that retrofitting security features to the legacy storage data path is more complex than we first expected.

Initially, we wanted to create a solution that would work with all popular file systems. However, we found the verification problem became much more tractable when working with file systems with proven consistency properties, as described in § 4.4.

Our earlier designs experimented with different methods to propagate information across storage layers, such as adding new special synchronous I/O requests and sending direct flash commands from the file system. After struggling to work against the asynchrony in the data path, we instead associated secure-deletion information with the legacy data path flow. We also decoupled the storage-specific secure-deletion action from the secure information propagation for ease of portability to different storage types.

We also found it tricky to design the GUID scheme due to in-transit versions and the placement of GUIDs. To illustrate, using only the sector number was insufficient when handling multiple in-transit updates to the same sector with conflicting sensitive statuses. Placing a GUID in transient data structures such as a block I/O structures led to complications when these structures could be split, concatenated, copied, and even destroyed before reaching storage. We solved this problem by associating a GUID with the specific memory pages that contain the data.

Tracking-granularity issues exist throughout the datapath. Data is stored in memory pages. File systems interact with blocks, multiples of which may exist on one memory page. The block layer may concatenate blocks together to form requests, which may span more than one memory page. Finally, requests are broken up into storage-specific granularities (e.g., flash pages). Metadata entries with mixed sensitive status can collocate within various access units as well. Various granularities make it difficult to map our solution to existing theoretical verification frameworks [34].

Finally, our work would not have been possible without direct access to a flash FTL. An unfortunate trend of FTLs is that they are mostly implemented in hardware, directly on the flash device controller. An implication is that most FTLs (and their wear-leveling/block-management routines) cannot be seen or accessed by the OS. To leave the door of software FTL research open, we need to create an environment that enables and eases experimentation, to demonstrate the benefits of software-level developments and controls.

To summarize, we have presented the design, implementation, evaluation, and verification of TrueErase, a legacy-compatible, per-file, secure-deletion framework that can stand alone or serve as a building block for encryption- and taint-based secure deletion solutions. We have identified and overcome the challenges of specifying and propagating information across storage layers. We show we can handle common system failures. We have verified TrueErase and its core logic via cases derived from file-system-consistency properties and state-space enumeration. Although a secure-deletion solution that can withstand diverse threats remains elusive, TrueErase is a promising step toward this goal.

11. ACKNOWLEDGMENTS

We thank Peter Reiher and anonymous reviewers for reviewing this paper. This work is sponsored by NSF CNS-0845672/CNS-1065127, DoE P200A060279, PEO, and FSU. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, DoE, PEO, or FSU.

12. REFERENCES

[1] Bauer, S. and Priyantha, N.B. 2001. Secure data deletion for Linux file systems. *Proceedings of the 10th Usenix Security Symposium* (2001), 153–164.

[2] Boneh, D. and Lipton, R. 1996. A revocable backup system. *USENIX Security Symposium* (1996), 91–96.

[3] Cooke, J. 2007. Flash memory technology direction. *Micron Applications Engineering Document*. (2007).

[4] CWE - CWE-327: Use of a Broken or Risky Cryptographic Algorithm (2.2): *http://cwe.mitre.org/data/definitions/327.html*. Accessed: 2012-09-05.

[5] Diesburg, S.M., Meyers, C.R., Lary, D.M. and Wang, A.I.A. 2008. When cryptography meets storage. *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability* (2008), 11–20.

[6] Ganger, G.R. 2001. *Blurring the line between OSes and storage devices*. Technical Report CMU-CS-01-166, Carnegie Mellon University.

[7] Garfinkel, S.L. and Shelat, A. 2003. Remembrance of data passed: a study of disk sanitization practices. *Security Privacy, IEEE*. 1, 1 (Feb. 2003), 17 – 27.

[8] Geambasu, R., Kohno, T., Levy, A.A. and Levy, H.M. 2009. Vanish: increasing data privacy with self-destructing data. *Proceedings of the 18th USENIX Security Symposium* (Berkeley, CA, USA, 2009), 299–316.

[9] Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J. and Felten, E.W. 2009. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*. 52, 5 (May. 2009), 91–98.

[10] Health Insurance Portability and Accountability Act of 1996: *http://www.hhs.gov/ocr/privacy/hipaa/administrative/statute/hipaastatutepdf.pdf*. Accessed: 2012-07-24.

[11] Hughes, G. 2004. *CMRR Protocols for disk drive secure erase*. Technical report, Center for Magnetic Recording Research, University of California, San Diego.

[12] Hughes, G.F. 2002. Wise drives [hard disk drive]. *Spectrum, IEEE*. 39, 8 (Aug. 2002), 37 – 41.

[13] Ironkey: *http://www.ironkey.com*. Accessed: 2012-07-26.

[14] Joukov, N., Papaxenopoulos, H. and Zadok, E. 2006. Secure deletion myths, issues, and solutions. *Proceedings of the Second ACM Workshop on Storage Security and Survivability* (New York, NY, USA, 2006), 61–66.

[15] Joukov, N. and Zadok, E. 2005. Adding secure deletion to your favorite file system. *Security in Storage Workshop, 2005. SISW '05. Third IEEE International* (Dec. 2005), 8 pp.–70.

[16] Katcher, J. 1997. *Postmark: A new file system benchmark*. Technical Report TR3022, Network Appliance, 1997. www. netapp. com/tech_library/3022. html.

[17] King, C. and Vidas, T. 2011. Empirical analysis of solid state disk data retention when used with contemporary operating systems. *Digital Investigation*. 8, (2011), S111–S117.

[18] Lee, J., Heo, J., Cho, Y., Hong, J. and Shin, S.Y. 2008. Secure deletion for NAND flash file system. *Proceedings of the 2008 ACM Symposium on Applied Computing* (New York, NY, USA, 2008), 1710–1714.

[19] Mac OS X Security Configuration for Mac OS X Version 10.6 Snow Leopard: *http://images.apple.com/support/security/guides/docs/SnowLeopard_Security_Config_v10.6.pdf*. Accessed: 2012-07-25.

[20] Marcel Breeuwsma, Martien De Jongh, Coert Klaver, Ronald Van Der Knijff and Roeloffs, M. 2009. *Forensic Data Recovery from Flash Memory*. CiteSeerX.

[21] National Industrial Security Program Operating Manual 5220.22-M: 1995. *http://www.usaid.gov/policy/ads/500/d522022m.pdf*. Accessed: 2012-07-26.

[22] Nightingale, E.B., Veeraraghavan, K., Chen, P.M. and Flinn, J. 2008. Rethink the sync. *ACM Trans. Comput. Syst.* 26, 3 (Sep. 2008), 6:1–6:26.

[23] OpenSSH: *http://openssh.com/*. Accessed: 2012-06-07.

[24] Perlman, R. 2005. *The ephemerizer: making data disappear*. Sun Microsystems, Inc.

[25] Peterson, Z.N.J., Burns, R., Herring, J., Stubblefield, A. and Rubin, A. 2005. Secure deletion for a versioning file system. *Proceedings of the USENIX Conference on File And Storage Technologies (FAST)* (2005), 143–154.

[26] Reardon, J., Capkun, S. and Basin, D. 2012. Data Node Encrypted File System: Efficient Secure Deletion for Flash Memory. *21st USENIX Security Symposium* (Aug. 2012).

[27] Reardon, J., Marforio, C., Capkun, S. and Basin, D. 2011. *Secure Deletion on Log-structured File Systems*. Technical Report arXiv:1106.0917.

[28] Scrub utility: *http://code.google.com/p/diskscrub/*. Accessed: 2012-07-26.

[29] Secure rm: *http://sourceforge.net/projects/srm/*. Accessed: 2012-07-26.

[30] Secure USB Flash Drives | Kingston: *http://www.kingston.com/us/usb/encrypted_security*. Accessed: 2012-07-26.

[31] shred(1) - Linux man page: *http://linux.die.net/man/1/shred*. Accessed: 2012-08-13.

[32] Shu, F. and Obr, N. 2007. *Data set management commands proposal for ATA8-ACS2*.

[33] Sivathanu, G., Sundararaman, S. and Zadok, E. 2006. Type-safe disks. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), 15–28.

[34] Sivathanu, M., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H. and Jha, S. 2005. A logic of file systems. *Proceedings of the 4th USENIX Conference on File and Storage Technologies - Volume 4* (Berkeley, CA, USA, 2005), 1–1.

[35] Sivathanu, M., Bairavasundaram, L.N., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H. 2004. Life or death at block-level. *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), 26–26.

[36] Sivathanu, M., Prabhakaran, V., Popovici, F.I., Denehy, T.E., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H. 2003. Semantically-smart disk systems. *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003), 73–88.

[37] Special Publication 800-88: Guidelines for Media Sanitization: 2006. *http://csrc.nist.gov/publications/nistpubs/800-88/NISTSP800-88_with-errata.pdf*. Accessed: 2012-07-26.

[38] Sun, K., Choi, J., Lee, D. and Noh, S.H. 2008. Models and Design of an Adaptive Hybrid Scheme for Secure Deletion of Data in Consumer Electronics. *Consumer Electronics, IEEE Transactions on*. 54, 1 (Feb. 2008), 100 –104.

[39] The OpenSSD Project: *http://www.openssd-project.org/wiki/The_OpenSSD_Project*. Accessed: 2012-07-29.

[40] Thibadeau, R. 2006. Trusted Computing for Disk Drives and Other Peripherals. *Security Privacy, IEEE*. 4, 5 (Oct. 2006), 26 –33.

[41] Wei, M., Grupp, L.M., Spada, F.E. and Swanson, S. 2011. Reliably erasing data from flash-based solid state drives. *Proceedings of the 9th USENIX Conference on File and Stroage Technologies* (Berkeley, CA, USA, 2011), 8–8.

[42] Wipe: Secure File Deletion: *http://wipe.sourceforge.net/*. Accessed: 2012-07-26.