TrueErase: Full-storage-data-path Per-file Secure Deletion

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 delete sensitive data securely from electronic storage is becoming an increasing concern. However, current per-file deletion solutions tend to be limited to one segment of the operating system's storage data path, and may leave behind sensitive data when interacting with storage components such as journaling, file-system caches, and certain storage media such as NAND flash.

This work introduces TrueErase, a secure-deletion framework. Through design, implementation, verification, and evaluation, we show that it is possible to build a legacy-compatible full-storage-data-path framework that performs per-file secure deletion, works with common file systems and emerging solid-state storage, while surviving common system failures.

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

The ability to ensure file deletion from digital storage is a pressing concern, as frequent incidences of data theft plague government [VE11b], academia [GO11; PS11], and industry [LE11]. Although wary users may be diligently emptying their digital trash bins and reformatting their storage, little do they know that typical deletion methods only make files invisible to users, while leaving behind recoverable sensitive data. As a consequence, data thought to have been erased may still be retrieved from a decommissioned storage device [GA03; JO09].

Secure deletion is concerned with rendering a file's removed content and metadata (e.g., name) irrecoverable. Achieving secure deletion is difficult due to diverse threat models. To gain a foothold to address this overarching problem, this paper focuses on dead forensics attacks on the local storage device, which occur after the computer has been shut down properly. Attacks on storage backups or live systems, cold boot attacks [HA08], covert channels, and policy violations are beyond our scope.

To simplify our design, we focus on the scenario where we have full control of the entire storage data path in a non-distributed environment. Thus, the basic research question is that under benign user control and system environments, what holistic solution can we design and implement to ensure that the secure deletion of a file is honored throughout the legacy storage data path?

We introduce TrueErase, a framework that irrevocably deletes data and metadata. Through design, implementation, verification, and evaluation, we have demonstrated that TrueErase is unique in the following combination of properties: (1) TrueErase works with legacy applications, common file systems, and emerging storage media (e.g., NAND flash). (2) Unlike techniques that involve the physical destruction of storage devices [DO95; BE04; OS11], TrueErase offers the flexibility to securely delete individual files (e.g., for cases such as expired client data, statute of limitations, etc.). (3) Unlike most per-file solutions (§2.1, §8), TrueErase is holistic and covers the entire storage data path, so that a secure-deletion operation issued by one segment of the data path (e.g., file system) will not be negated by another (e.g., flash that keeps versions of data). (4) TrueErase is designed to survive common system failures, and (5) the core logic of the framework is verified systematically.

User space	Applications
Kernel space	VFS
	File system
	Block-layer interface
Kernel/hardware	Storage-management layer
Hardware	Storage

Figure 2.1. Legacy storage data path.

2. Existing Approaches and Challenges

Typical file deletion removes only the i-node or a file's metadata, with the content left on the storage medium until it is overwritten. Even recreating the file system does not ensure secure deletion. For example, the MSDOS `format` command overwrites 0.11% of the data [GA03].

Various solutions have been proposed; however, most of them operate at one segment or level/layer, if viewed vertically on the legacy data path (Figure 2.1). While layering encourages the portability of individual layers, the types of information available at each layer restrict the effectiveness of secure-deletion solutions.

2.1. Existing approaches

Storage-management-level solutions: At the lowest level, the storage-management layer handles vendor-specific representations on the storage media. Since this layer accepts requests from layers above through the block-layer interface in terms of logical blocks, it has no information

on a block's type (e.g., data, metadata), file ownership, in-use status, etc. [GA01; SI04].

Subsequently, solutions at this layer tend to operate on all blocks indiscriminately, with no support for per-file secure deletion. An example is encrypting the whole device, and then performing secure deletion via revoking the master key [HU02; TH06; NI06; KI08; SA08; HU09; IR11; VE11a]. Note that even if a storage device offers secure deletion to selective blocks (e.g., per-block sanitization [WE11]), it needs the ability to map blocks to files and metadata to support secure deletion at the file granularity [KI11].

File-system-level solutions: File-system solutions reside in the virtual file system (VFS) [KL86] and the file-system layers [JO05; PE05; JO06]. (The VFS implements common file-system functions such as caching and allows various file systems to run simultaneously.)

The file-system layer generally is unaware of the storage medium and has no control over the physical storage location of data and metadata. Thus, a file system may issue writes of random bits to the same location, intending to perform military-grade secure deletion [NI06] via *in-place* disk overwrites. Instead, the underlying storage medium can actually be a flash drive. Such devices typically write every version of an update (even to the same logical location) to a different physical location, to avoid the slow erases required by flash before updating the same location, negating the intent of secure deletion. Note that per-file, encryption-based solutions will not solve the problem, since the lower layer may not honor the secure-deletion semantics to remove file keys.

File-system-layer solutions may also overlook the sensitive information left behind by common journaling mechanisms, which log information to ease recovery.

User-space solutions: At the application layer, solutions are carried out by user-level programs. Given that these programs operate through file systems, they inherit the same constraints. Additionally, user programs [KO07; NE09; GA10; SM11; PL11; YO11] have limited control of a file's metadata, and thus cannot enforce secure deletion, leading to the leakage of information such as file names, file sizes, etc.

Cross-layer solutions: Cross-layer secure-deletion solutions do exist. However, some are not built for the purpose of secure deletion, and do not interact well with the legacy storage data path [SI03; SH07]. Ones that are designed for secure deletion are tailored for specific file systems or storage media [SI06; LE08; SU08] (see §8).

2.2. Other secure-deletion challenges

There are other secure-deletion challenges as well:

No legacy requests to delete data content: 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. Even issuing zero-filled data-block requests to the same logical location cannot achieve secure deletion, as discussed in §2.1.

Complex storage-data-path optimizations: Secure-deletion operations need to be compatible with legacy optimizations. In particular, storage requests may be reordered, merged (concatenated), split, consolidated (applying one update instead of many to the same location), cancelled, or buffered (e.g., journaled) with versions of requests in transit.

Lack of data-path-wide identification: Tracking request versions across storage layers is complicated by the possible reuse of runtime data structures, identification numbers (e.g., i-node), and memory addresses.

System crashes: Our proposed secure-deletion solution must work with common journal-recovery mechanisms, and our persistent states need to survive system crashes.

Verification: While verification is often overlooked by various deletion systems, we need to ensure that (1) secure deletions are correctly propagated throughout the storage data path and that (2) various assumptions are checked whenever possible.

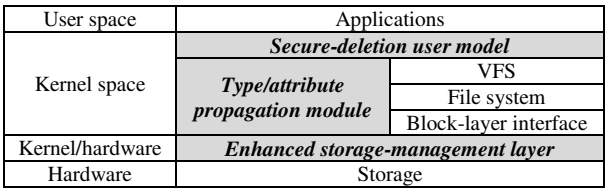

Figure 3.1. TrueErase framework (shaded boxes).

3. TrueErase Overview

We introduce TrueErase (shaded boxes in Figure 3.1), a framework that propagates file-level information to the block-level storage-management layer, so that per-file secure deletion can be honored throughout the data path.

Secure-deletion user model: A user can use legacy attribute-setting tools to specify files and directories whose data and metadata will be securely deleted. Unmodified legacy applications can operate on those objects.

Type/attribute propagation module: The VFS and file system layers can report information and reminders of pending update events (e.g., zeroing bitmaps) to a centralized, type/attribute propagation (***TAP***) module. We use a global unique-ID scheme to track versions of in-transit storage requests. To handle legacy optimizations, TAP only passively gathers and forwards information, and it does not alter the flow and ordering of update requests. TAP tracks only transient soft states. Thus, TrueErase needs no mechanisms to recover its states. During crash recovery, all operations replayed and reissued by a typical file system's journal-recovery procedure are handled securely. All remnants of data and metadata in the storage data path from the prior session are securely deleted.

Implicitly extended block-layer interface: The block-layer interface is, in a sense, extended, so that the storage-management layer can inquire TAP about file-level

information prior to performing secure operations on a block based on its type(s) (e.g., data, i-node) and attributes.

When the block status is unknown to TAP (e.g., a missing report from a file system), the block will be handled securely when removed. (This behavior is insufficient to achieve secure deletion because typical file systems do not issue deletion requests for file content.)

Enhanced storage-management layer: We have added secure-deletion commands for different storage media to show that our solution generalizes. Due to length restrictions, most of our descriptions in this paper are confined to the flash medium.

Certain optimizations are disabled when handling secure-deletion operations (e.g., storage-built-in caches).

Verification tools: We provide kernel- and user-level tools to detect illegal changes in the block type, incorrect triggering of deletion events, and missing reports from the VFS and file-system layers. Additionally, we provide tools to verify the core system states and transitions.

Assumptions: Because of the complexity of the storage data path, in this work, we assume a benign personal computing environment (e.g., laptops), where a user has the administrative control of an uncompromised, single-user, single-file-system, non-RAID, and non-distributed system. The threat model is the *dead forensics* attacks, which occur after a user unmounts and shuts down the system after completing secure-deletion operations.

At the system level, we assume control of the entire storage data path, including the storage-management layer.

Because our work focuses on the correct propagation of information across storage layers to ensure secure deletion, our solution will not propagate the sensitive status of a file to another file (e.g., via tainting) or handle copies/references of files made by users or applications.

Also, we assume the use of common journaling file systems that adhere to the integrity properties specified by [SI05] (e.g., ext3). Further, all update events and block types are reported to our framework. These assumptions allow us to concentrate on building and verifying the properties of secure deletion.

4. TrueErase Design

The major areas of TrueErase's design include (1) specifying which files or directories are to be securely deleted via a user model, (2) tracking and propagating information across storage layers via TAP, (3) enforcing secure deletion via storage-medium-matching mechanisms added to the storage-management layer, and (4) exploiting file-system-integrity properties to identify and handle corner cases of secure deletion.

4.1. User model

Having a per-file secure-deletion capability is essential when deleting the entire device is not an option. Also, deleting all files securely can be prohibitive in terms of the performance overhead.

Ideally, a user can follow the traditional permission semantics and apply common extended-attribute-setting tools to mark a file or directory as ***sensitive***, which means it will be securely deleted when removed. A legacy application then can issue normal operations that translate into secure-deletion operations on those specified files or directories to remove sensitive file data content, metadata (e.g., i-node, file name, etc.), and associated copies within the storage data path.

Toggling the sensitive status: However, certain secure-deletion semantics cannot easily follow traditional permission models. For example, toggling the sensitive status is different from toggling read/write access controls. To illustrate, prior to a file or directory being marked sensitive, older versions of data and metadata may have already been left behind. Without tracking all file versions or removing old versions for all files at all times, TrueErase enforces secure deletion only for files or directories that have stayed sensitive since their creation. Should a non-sensitive file be marked sensitive, secure deletion will be carried out on a best-effort basis (indicated by an extended-attribute flag). That is, only versions of metadata and file content after the file is marked sensitive will be securely deleted.

Name handling: Another semantic deviation occurs when secure deleting a file name. A directory is typically represented as a special file, with its data content storing the file names it holds. Although the permissions of a file are applied to its data content, permission to handle its name is controlled by its parent directory. However, under TrueErase, the sensitive status of a file is applied to its name as well. Thus, marking a file sensitive will also cause some of its parent directory's data content to be securely handled, even if the parent directory is not marked sensitive. Without this partial marking, marking a file sensitive would bubble the sensitive status from its parent to the root directory, negating our per-file solution goal.

One deployment issue is that by the time a user can set attributes on a file, its name may already be stored non-sensitively. Without modifying VFS, one remedy is to create files or directories under a sensitive directory so they can inherit the attribute implicitly. To ensure that the name of the top-level sensitive directory will be securely deleted, a user can use our `smkdir` wrapper script, which first creates a directory with a temporary name, marks it sensitive, and renames it to the sensitive name.

Links: The secure-deletion semantics for hard links and symbolic links follow traditional permission semantics. A hard link to a sensitive file will perform secure deletion when the link count drops to zero. A symbolic link's own permissions govern whether its name and content (path to a target) will be securely deleted, independent of the object being referenced.

4.2. TAP module

The TAP kernel module tracks and propagates information from the VFS and file-system layers to the storage-management layer.

What and how to track? We had to solve many issues with tracking and propagating information.

Where to instantiate requests to delete file content: We motivate our deletion-tracking design through the disadvantage of block zeroing for all types of storage. Given that TAP can propagate information across layers, a file system can simply send blocks of zeros to TAP with annotations, and the storage-management layer can tell which blocks need to be deleted securely. However, for storage devices such as NAND flash, a previously written location needs to be explicitly erased (resetting all bits to 1s or 0s) via an erase command before being written again. Overwriting a location with zeros would trigger both an erase command and a write of zeros. Also, a subsequent update to the same location would again trigger an erase before the update is written.

Instead, TAP allows file systems to attach compact reminders (e.g., deleting blocks that hold the file content) to other secure-deletion requests (e.g., setting the allocation bits to zeros). The storage-management layer can choose a secure-deletion method that matches the underlying medium. For the NAND flash case, triggering the erase command once may be sufficient (details in §4.3).

Tracking deletion is not enough: By the time a secure-deletion operation is issued for a file, versions of its blocks may have been created and stored (e.g., due to flash optimizations), and the current metadata may not reference old versions. One approach is to track all versions, so that they can be deleted at secure-deletion time. This approach potentially allows sensitive updates to the same blocks to be consolidated. However, tracking these versions requires persistent states and mechanisms for those states to stay consistent across system failures.

Instead, TrueErase deletes old versions along the way to avoid extra persistent states. That is, the secure-deletion mechanism is triggered for each update that intends to overwrite a sensitive data or metadata block in place (***secure write*** for short). Therefore, in addition to deletion operations, TrueErase needs to track all in-transit updates of sensitive blocks.

Tracking sensitive updates is still not enough: Given the small size of per-file metadata, a metadata block often stores metadata for many files, with possibly mixed sensitive status. Thus, when a non-sensitive file shares a metadata block with a sensitive one, updating non-sensitive metadata may also cause the sensitive metadata to appear in the storage data path. Thus, any shared metadata blocks with a mixed sensitive status will be treated sensitively.

Global unique page IDs: Given the possible reuse of data structures, namespace, and memory addresses within the storage data path, we added a monotonically increasing globally unique page ID (i.e., reboot epoch number concatenated with a counter) to each memory page at its allocation time, so that even the same page reallocated to hold versions of the same logical block will have different page IDs.

Sector-based tracking granularity: Many different tracking units exist in the data path, such as logical blocks for file systems, concatenated requests, physical sectors, and device-specific units. TAP tracks by physical sector, because it is unique to the storage device and can be computed from anywhere in the storage data path. The globally unique page ID and the physical sector number obtained by TAP from lower layers form a global unique ID (***GUID***), which we can use to uniquely associate attribute propagation with the smallest applicable secure-deletion unit in the storage data path.

How to interact with TAP? Table 4.2.1 summarizes the TAP interface. Section 5 describes its usage.

Table 4.2.1. TAP interface.

Core interface
`Report_write()`: A file system can ask TAP to create per-physical-sector ***write entries*** (or update them if they already exist) to track updates to a block. To do so, a file system needs to specify the globally unique page ID of the logical block, its type (e.g. i-node, data), its sensitive status, and logical sector numbers that will be translated to physical sector numbers and tracked.
`Report_delete()`: A file system can ask TAP to create reminders for pending data deletes for particular sectors. These reminders will be attached to a write entry with a specified GUID. A file system can also specify whether the reminder actions should be performed before/after the piggybacked request.
`Report_copy()`: The file-system layer can inform TAP via an origin-destination GUID pair when a memory copy of a block is made. Reminders associated with sectors within the original block are transferred to the corresponding sectors of the destination block.
`Cleanup_write()`: A file system can inform TAP to remove the write entry with a specified GUID. This call also handles the scenario when a file has already been created, written, and deleted before the VFS cache has a chance to flush.
`Check_info()`: The block layer is unchanged. However, when the storage-management layer receives a request, it can query TAP via this call to retrieve information about the request with its GUID. The retrieved information will indicate the sensitive status of a storage update and whether deletion reminder requests should be carried out before or after the update.
Derived interface
`Report_write_journal()` is a special case of `Report_copy()`.
`Cleanup_journal()` is a special case of `Report_delete()`.
`Erase_journal()` is a special case of `Report_delete()`, which erases the entire journal bounds on mount.

4.3. Enhanced storage-management layer

TrueErase does not choose the secure-deletion mechanisms until a storage request has reached the storage-management layer. By doing so, we can be assured that the chosen mechanisms match the characteristics of the underlying storage medium. We used a NAND flash storage-management component for this demonstration.

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

[CO07]; (2) NAND reads and writes are in ***flash pages*** (e.g., 2-8 Kbytes), but erasures are performed in ***flash blocks*** (e.g., 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 during this process—and (4) each storage location can be erased only about 10K-1M times [CO07].

As a common optimization, when flash receives a request to overwrite a flash page, the ***flash translation layer (FTL)*** remaps the write to a pre-erased flash page (with a version stamp) and marks the old flash 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, but can be recovered by forensic techniques [BR07]. To prolong the lifespan of the flash, ***wear-leveling*** techniques are often used to evenly spread the number of erasures across all storage locations.

NAND secure commands: We added two secure-deletion commands to the storage-management layer for NAND flash:

`Secure_delete()`: This command pertains to a group of pages in a single flash block we would like to secure delete. The command copies other in-use, not-to-be-deleted pages from the current flash block to other areas as new page versions, while marking the old versions as unused. After all other in-use pages have been migrated, the target page can be marked invalid, and the current flash block can be erased via the flash erase command.

`Secure_write()`: Our secure-write command allows the specified page to be written first before executing `Secure_delete()` on the old page (if any).

Storage-management-layer crash handling: When a crash occurs, it is possible that only some of the in-use sensitive pages within a flash block have been copied elsewhere. Since copies are made before erasing the old versions, we do not need to worry about data loss. However, other sensitive pages on the same flash block may now have duplicates. Given that the secure deletion of the page did not complete, the common journal crash-recovery mechanism will reissue the operation, so that other remaining in-use pages in the same flash block can continue the migration, and the block can then be erased.

Wear leveling: When a NAND flash 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 respect for file boundaries, sensitive status, etc. Thus, in addition to storing a file's sensitive status in the extended attribute, we store a sensitive-status bit in the per-page ***control area***. (This area also contains checksum and 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()`.

Note that this bit is primarily for handling internal device traffic and may not always be in sync with the latest file-system-level sensitive status. For example, data blocks of short-lived non-sensitive files may never reach persistent storage. Thus, whenever a physical page's sensitive status disagrees with the sensitive status of an incoming request, we will mark the page sensitive and treat it as such.

4.4. File-system-integrity properties and secure deletion

By working with file systems that adhere to the integrity properties defined by [SI05], TrueErase can leverage cases that are protected by those properties and use those properties to flush out corner cases.

In a grossly simplified sense, as long as pieces of file metadata reference the correct data and metadata versions throughout the storage data path, the system is considered consistent. In particular, we are interested in three properties in [SI05]. The first two are for non-journaling-based file systems. Without holding both properties (e.g., ext2), a non-sensitive file may end up with data blocks from a sensitive file after a crash recovery. The last one is needed for journaling-based file systems.

(1) 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. If violated, a reused data block may become persistent before the previous block's file ownership is updated persistently. A crash in between will result in the old file owning the data block of a new file. Then, a secure deletion of the old file would result in deleting file content of the new one.

Ensuring persistence: We need to disable storage-built-in write caches or use barriers and device-specific flush calls to ensure that persistence of an update is achieved.

Static file ownerships and types for in-transit blocks: In the secure-deletion context, 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 these guarantees, we do not need to worry about the possibility of dynamic file ownerships and types for in-transit blocks.

Block versions in transit: On the other hand, this property does not mention the uniqueness of the block in the storage data path, and we still need to use GUIDs for tracking versions of the block.

Dynamic sensitive-mode changes for in-transit blocks: Also, the block may change its sensitive status across the data path. To simplify tracking and the handling of the sensitive status of a block, we allow only a non-sensitive in-transit file or directory to be marked sensitive, but a sensitive object is not allowed to be marked non-sensitive.

(2) The ***pointer-ordering property*** ensures that a referenced data block in memory will become persistent before the metadata block in memory that references it. With reversed ordering, a system crash could cause the persistent metadata block to point to a persistent data block

location not yet written. Should this location be allocated to a new file, a secure deletion of the old file would secure delete the content of the new file.

Secure-deletion point for data blocks: For a data block that is no longer referenced by its metadata block during a normal deletion, this property does not specify a need to make the data block persistent. This property allows various file systems to omit the data-content deletion while not compromising the integrity.

If we want end users to see secure deletion as updating a data block with zeros, this update needs to become persistent before the data block's metadata block that references it becomes persistent. To be specific, deleting file content typically involves the following (simplified) steps: (t1) updating an i-node to set the file size to zero, (t2) updating metadata blocks (e.g., indirect blocks) to remove pointers to data blocks, and (t3) updating the bitmap allocation blocks to indicate the availability of blocks for reuse. Based on the reuse-reordering property, the last point at which secure deletion of data blocks can be inserted is right before step (t3). However, because we want users to see a partially securely deleted file filled with zeros after a crash (in a way that is consistent with the file-integrity constraint), we have inserted (t0) secure deletion of data blocks just prior to step (t1).

Modified data blocks that are no longer referenced: The pointer-ordering property also does not specify the fate of updated data blocks in memory once references to them are removed. One concern is that these unreferenced updated data blocks can result in writes that arrive after we have securely wiped the same data blocks. Currently, the legacy VFS prohibits such writes from occurring.

Crash after a sensitive data block becomes persistent: The pointer-ordering property further indicates that right after a newly allocated sensitive data block becomes persistent, a crash at this point will result in the block being unreferenced by its file. To address this concern, we will perform secure deletion on unreferenced sensitive blocks at recovery time (see §4.5).

Secure-deletion point for metadata blocks: For metadata blocks that behave like data blocks (e.g., directory content), the deletion point is the same as that of data blocks. The remaining metadata blocks are securely updated or deleted as they reach the storage (see §5.2.2).

(3) The ***non-rollback property*** ensures that older versions of data or metadata blocks will not overwrite newer versions persistently. This property is important for journaling file systems with versions of storage requests in transit. If we issue a secure deletion to a block after an update to the same block, we do not need to worry about the two affecting the persistent storage in the wrong order from the file system's standpoint. However, we need to worry about reordering at lower layers. We further need to handle the possibility of consolidating, merging, and splitting storage requests.

Reordering of requests: A storage device can contain a built-in cache and determine its own ordering to make updates persistent. Thus, we need to disable storage-built-in write caches or use barriers and device-specific flush calls to ensure that proper orders are enforced.

Consolidation of sensitive and non-sensitive requests: Certain cases of consolidation are disabled (e.g., repeated writes of random bits to the same location on disk). When permitted (such as in the page cache or journal), we make conservative interpretations of an update's sensitive status when consolidating updates. Basically, as long as one of the updates to a given location is sensitive, the resulting update will be sensitive.

Merging/splitting of sensitive/non-sensitive requests: Because TAP tracks the sensitive status at the sector granularity, concatenating and splitting at the request level will not affect our tracking and performing secure deletion.

Summary of secure-deletion cases: With file-system-integrity properties, we can see the structure of secure-deletion cases and handle them: (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 referenced by file-system metadata persistently, (3) making sure that secure deletion is not applied to the wrong file, (4) making sure that a securely deleted block is not overwritten by a buffered write from a file that no longer exists, and (5) handling versions of a storage request in transit (mode changing, reordering, consolidation, merging, and splitting). Buffering, asynchrony, and cancelling of requests are handled by TAP.

4.5. Miscellaneous design points

Crash handling: Persistent states at the file-system level are handled by extended attributes and are protected by journal-recovery mechanisms. TAP contains no persistent states and requires no mechanisms for recovery. The sensitive bits in persistent storage are loosely synchronized with the file-system layer via the mechanisms in §4.3.

Thus, at recovery time, the journal is replayed with all operations handled securely. We then securely delete the entire journal. To hunt down leftover sensitive data blocks, for flash, we securely delete sensitive blocks that are not marked allocated by the file system. For disks, we sequentially overwrite all free space with random bits.

Swapping and hibernation: Swapping and hibernation are disabled in our current system; handling these features is future work.

5. TrueErase Implementation

We prototyped TrueErase under Linux 2.6.25.6. We chose ext3 and its jbd journaling layer due to their popularity and adherence to file-system integrity properties [SI05]. 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) Linux kernel module as our FTL. Although DiskonChip is somewhat dated, our design is applicable to modern flash and development environments.

In terms of the development effort, our user model required 269 lines of C code; TAP, 939; secure-deletion commands for flash, 592; user-level development environment for kernel code, 1,831; and verification framework, 8,578.

5.1. Extended attributes

We use Linux's extended attributes to record which files need secure erasure. A user can mark a file or directory as sensitive by using the `setfattr` command to set its `TrueErase.security` attribute, and can check the sensitive status with `getfattr`.

5.2. TAP module

TAP is implemented as a kernel module that tracks all update and secure-deletion requests via the interface mentioned in §4.2. We inserted 60 TAP-reporting calls in ext3 and jbd, with most of them collocated with `submit_bh()` and various dirty functions (e.g., `ext3_journal_dirty_data`). 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 deletion under ext3: Ext3 deletes the data content of a file via its truncate function, which involves steps (t1) to (t3) mentioned in §4.4. Multiple rounds of truncates may be required for deleting the data content of a large file.

Deleting a file involves these steps: (d1) removing the name and i-node reference from the directory, (d2) adding the removed i-node to an orphan list, (d3) truncating the entire file via (t1) to (t3), (d4) removing the i-node from the orphan list, (d5) removing the extended attributes, and (d6) updating the i-node map to free the i-node.

Journaling: Typical journaling employs the notion of a ***transaction***, so that either the entire group of writes or none of the writes make it to their final storage locations. 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 before (j2) they are propagated to their final storage destinations. (j3) A committed and propagated transaction then 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, the block can be used by another file according to the reuse-ordering property.

At recovery time, committed transactions in the journal are replayed to re-propagate or continue propagating the changes to their final destinations. Uncommitted or halfway written transactions in the journal are aborted, without their effects reflected in the final storage destinations.

Jbd: Jbd differentiates file data and metadata (i.e., everything else). We chose the commonly used ordered mode, which journals only metadata but requires (j0) a data block to be propagated to the final destination before its metadata blocks are committed to the journal.

5.2.2. Tracking secure-write and -deletion operations

Because all truncation, file deletion, and journaling steps can be expressed and performed as secure writes and deletions to data and metadata blocks, we will illustrate only how these operations are tracked by TAP.

Applicable block types: We perform secure writes and deletions to sensitive data blocks, i-node blocks, extended-attribute blocks, indirect blocks, directory blocks, and those corresponding structures written to the journal. The remaining metadata blocks are frequently updated (e.g., superblocks) 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 5.2.2.1): Ext3/jbd calls `Report_write()` on sensitive data block updates, and TAP creates per-sector write entries based on GUIDs. Under TAP, updates to the same logical block are consolidated via GUIDs; this behavior matches that of the page cache.

The data update eventually reaches the storage-management layer (via ext3 commit), which retrieves the sensitive status from TAP via `Check_info()`. The layer then can perform the secure-write operation that matches the underlying storage medium, followed by invoking `Cleanup_write()` to remove the corresponding write entries.

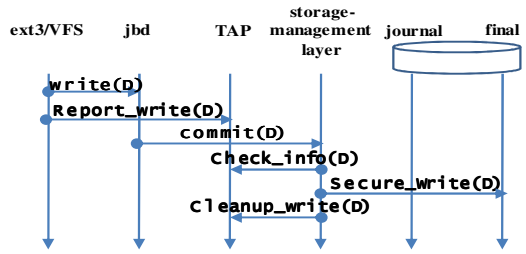

Figure 5.2.2.1. Secure data updates. D is the reported data block.

Secure metadata updates: A metadata block is journaled by jbd in ordered mode. Thus, in addition to writing it securely to its final storage destination, it must be securely written to and deleted from the journal.

Ext3/jbd also reports metadata updates to TAP via `Report_write()`. Repeated updates to the same metadata entries will lead to updates to TAP write entries with matching GUIDs. To commit a transaction, jbd calls

`Report_write_journal()` to clone write entries and their sensitive status, with request destinations altered from final storage destinations to persistent journal locations. When the storage-management layer receives the journal update request via a journal commit, it performs checking, invokes the secure-write method, and cleans up the cloned journal write entries via `Cleanup_write()`, such as in Figure 5.2.2.2 (a).

The original write entries remain until they are propagated to their final storage destinations via a journal checkpoint. The storage-management layer performs checking, securely writes the metadata to the final destination, and cleans up the corresponding write entries, as shown in Figure 5.2.2.2 (b).

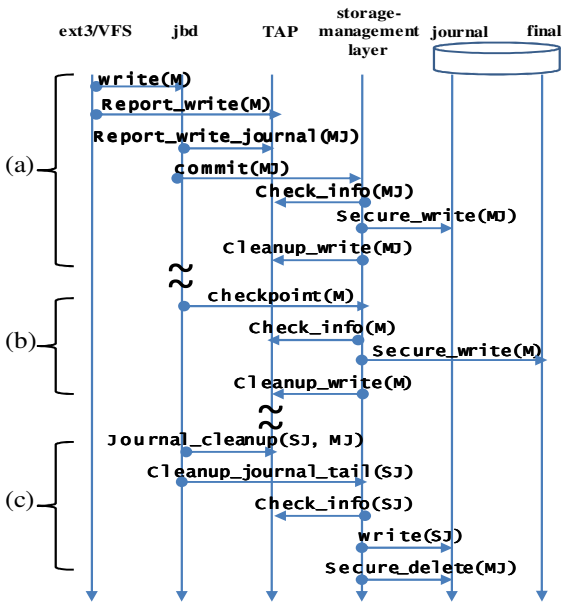

Figure 5.2.2.2. Secure metadata update. M is the reported metadata block; MJ is the reported metadata journal block; and SJ is the reported journal superblock. Steps (a), (b), and (c) correspond to steps (j1), (j2), and (j3) in §5.2.1.

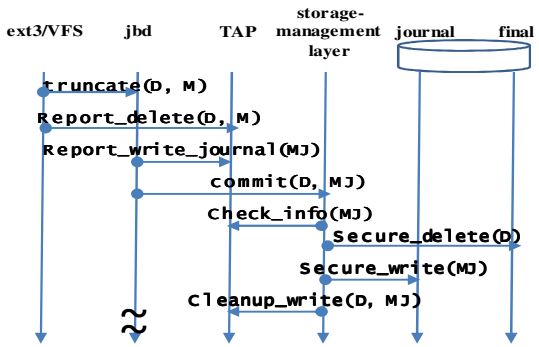

Figure 5.2.2.3. Secure data deletion. M is the reported metadata block; D is the reported data block; MJ is the reported metadata journal block. Events from Figure 5.2.2.2(b) and (c) occur at the end of this diagram (omitted for space).

Jbd tracks in-use persistent journal locations through its own superblock. Periodically, jbd checks the in-use status of journal locations in its log via `cleanup_journal_tail()`. If some of its log is no longer needed, it updates its superblock allocation pointers accordingly. We leverage this function to report journal locations no longer in use to TAP. If any of the locations match our write entries, we securely delete the location after the updated journal superblock is written, as shown in Figure 5.2.2.2 (c). In the case of a crash, we securely delete all usable journal log locations through `Erase_journal()` once transactions have been securely replayed (not shown in the figure)..

Beyond `cleanup_journal_tail()`, immediate secure deletion of journal entries is rather implementation specific. Thus, we currently use file-system unmounts to trigger immediate secure deletion of the entire journal.

Secure data deletions: When ext3 wishes to delete the contents of a sensitive file via its `truncate` function, it uses `Report_delete()` to inform TAP of blocks to be deleted and an i-node containing the updated file size. TAP will create secure-deletion reminders for those blocks attached to the write entry referring to that i-node. When the i-node write entry is copied via `Report_write_journal()`, reminders are transferred to the copy to ensure that secure deletions are applied to the matching instance of the i-node update.

When the storage-management layer receives the request to commit the update of the sensitive i-node to the journal, it will call `Check_info()` and retrieve the sensitive status of the i-node from TAP, along with locations to be securely deleted before committing the i-node update. The data areas are then securely deleted before the journaled i-node update is securely written to the journal on storage. The storage-management layer then cleans up the corresponding write entries, as shown in Figure 5.2.2.2 (b) and (c).

Note that before committing the journaled i-node update, all pending updates to the data blocks to be deleted have been propagated to their final storage destinations because of the pointer-ordering property. Thus, we do not need to worry about pending updates that will undo our secure deletions to these blocks.

Ext3/jbd upholds reuse ordering by copying the allocation bitmap before the deletion into a `b_committed_data` field associated with the deletion transaction. Until the modified bitmap (and file i-node) is committed, the file system is presented with the old copy of the bitmap. Thus, within a transaction, we do not have ambiguity in terms of the file ownership of a block when applying secure deletion.

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

Secure metadata deletions: During a file truncation or deletion, ext3 also deallocates the following metadata blocks: 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 `Report_write()`. Additionally, we need to zero out the i-node and variable-length file name in the memory copies, so that they will not negate the secure write performed at the storage-management layer.

Miscellaneous cases: When securely writing metadata, sometimes, an update to a metadata entry occurs in a new transaction before an old transaction containing the same entry has been committed. Because modifying blocks during a journal commit could yield system corruption, jbd will create a frozen copy for the old transaction. When this happens, jbd calls `Report_copy()` so that TAP will clone write entries for the frozen copy. Any delete reminders are transferred to the cloned write entry. When jbd commits the old transaction, it calls `Report_write_journal()` to commit write entries created for the frozen copy.

Another issue is that under jbd, committed transactions are not propagated immediately to their final destinations. Across committed transactions, the same metadata entry (e.g., i-node) might have changed its file ownership and sensitive status. Thus, jbd may consolidate a non-sensitive update 1, sensitive update 2, and non-sensitive update 3 to the same location into a non-sensitive update. To resolve this issue, once a TAP write entry is marked sensitive, it remains sensitive until it is securely written.

5.3. Enhanced storage-management layer

We used the INFTL Linux kernel module as our FTL.

Default INFTL behavior: 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 before it can be written.

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 stacked on the 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.

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

INFTL reads: For a read, INFTL traverses from the top of a stack to the bottom 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 0s.

INFTL extension: We added two commands.

Secure write: A secure write is similar to the current INFTL in-place update. However, if a stack contains a sensitive page, we set the maximum depth of the stack to 1 (0 is the stack top). Once the maximum is reached, the stack must be consolidated to depth 0. When consolidating, instead of leaving old blocks, they are immediately erased via the flash erase command.

Secure delete: A secure delete is a special case of 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.

Optimizations: To improve performance and reduce flash wear, we aligned the logical block boundaries with flash block boundaries. The TAP deletion reminders are grouped by flash blocks. We then can use this information to securely delete multiple pages on a block with only one round of migration for other in-use pages.

5.4 Disabled storage-management-layer optimizations

Since jbd waits for writes to reach storage before continuing between steps (j0), (j1), and (j2), other than storage-built-in caches, lower-layer reordering cannot reorder requests between steps and violate file system constraints. Thus, we used the no-op elevator scheduler. Our flash has no built-in cache to be disabled.

6. Verification

Our verification efforts include (1) testing the basic cases, assumptions, and corner cases in §4.4 and (2) verifying the state space of TAP. Although this level of verification exceeds that used for most other secure-deletion solutions, we plan further verification as future work.

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 both sensitive and non-sensitive objects; when the operation included both a source and a destination (or target) we tested all four possible combinations. The operations we checked included moving objects to new directories, replacing objects, and making and updating both symbolic and hard links. We also tested sparse files. In all cases, we verified that the operations behaved as expected.

PostMark: We ran the PostMark benchmark [KA97] with default settings, modified with 20% of the files

marked sensitive, with random content. Afterwards, we found no remnants of the sensitive information.

Reporting of all updates: In order to check that all update events and block types are reported, we looked for errors in the form of unanticipated block-type changes and unfound write entries in TAP, etc., which are signs of missing reports from the file system or the VFS layer. Currently, 100% of updates are reported.

Corner cases related to file-system integrity properties: For cases derived from the reuse-ordering property, we created a mini file system with most of its i-nodes and blocks locked down (allocated) to encourage reuse. Then we performed tight append/truncate and file creation/deletion loops with alternating sensitive status. We used predetermined random file content to detect sensitive information leaks and found none.

For cases derived from the pointer-ordering property, we verified our ability to recover from basic failures and remove remnants of sensitive information. We also verified that VFS prohibits unreferenced data blocks from being written to the storage. Performing fault injections at stages of truncation, deletion, and journaling will be future work.

Since the page ID component of GUIDs increases monotonically, we can use this 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, sensitive/non-sensitive/sensitive, etc.).

6.2. TAP verification

State representation: We exploited the properties of our system to trim the state space. (1) A write entry once created at report time will not consolidate with other write entries until it is removed. This property is necessary to assure 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. (2) The next state transition is based on current write entries of different types. With those two properties, we can reduce the representation of a state to having at most one write entry of each type, and explore all state-generating rules.

To illustrate, each state holds one write entry for nine types of blocks: 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. 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 specified input parameters. For example, the first `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 the `Report_write()` is called again to mark the i-node sensitive, S1 is transitioned to a new state S2, with both the allocated and sensitive bits set to 1s.

State-space enumeration: To enumerate states and transitions, we permute all TAP interface functions with all possible input parameters to the same set of write entries. Given that the enumeration step can be viewed as traversing a state-space tree in the 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 reachable states (starting with the 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. Of these, 61% are error transitions; 25% leading to creating two write entries of the same type, which is the boundary of our state representation; 5% are self-transitions; and 9% are legal transitions to other states.

Two-version-programming verification: 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. We identified and repaired four incorrect rules and three implementation bugs.

Table 7.1. Per-file-operation elapsed times/number of flash operations under different PostMark settings.

	elapsed times (secs)	page reads/writes		
		control-area reads/writes		erases
base	0.017	55/6	7/6	0.10
0% sensitive files	0.019	63/6	8/6	0.11
1%	0.057	120/19	33/18	0.32
5%	0.13	220/41	80/38	0.73
10%	0.17	280/55	110/51	1.0

7. Empirical Evaluation

We compared TrueErase to the unmodified Linux 2.6.25.6 system running ext3 locally. All reported numbers are based on 5 runs of experiments. The 90% confidence intervals are within 22% and are omitted for clarity. In terms of workloads, we used PostMark [KA97] to measure the overhead for metadata-intensive small-file I/Os. In terms of hardware/software settings, we also compiled OpenSSH [2011] version 5.1p1 to measure the usage for larger files. We conducted 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, running Linux 2.6.25.6.

PostMark: For each run, we used the default configuration except the following: 10K files, 10K transactions, 1-KB block size for reads and writes, and a read bias of 80%. We also modified PostMark to create

different percentages of files (chosen randomly) marked sensitive. Before running tests for each experimental setting, we dirtied our flash by running PostMark with 0% sensitive files just 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 7.1 shows that when TrueErase operates with no sensitive files, tracking and increased querying of extended attributes and metadata account for 10% overhead compared to the base case. With 1% of files marked sensitive, an average file operation takes 0.057 seconds, and this overhead grows sub-linearly since a metadata block update may consolidate metadata updates for multiple sensitive files. With 10% of files being sensitive, an average file operation can take 0.17 seconds, which is acceptable for interactive use. The high number of page reads and writes reflects the fact a flash block consists of 32 pages and in-use pages need to be migrated during secure operations.

OpenSSH compilations: We issued `make` + `sync` to measure the elapsed times for compiling OpenSSH. For the TrueErase case, we marked the `openbsd-compat` directory sensitive before issuing `make`. Only newly created files in that directory are updated and deleted securely, accounting for roughly 27% of the newly generated files. Before running each set of tests, we dirtied the flash the same way as we ran PostMark.

Our results show that a user would experience a slowdown within a factor of two. This overhead reflects the cost of flash erasure normally hidden by asynchrony. It also highlights the dated nature of the INFTL page and block allocation policy. Optimizing flash layout and developing deletion-time (as opposed to per-update) secure deletion mechanisms will be areas of future work.

8. Related Work

This section will discuss only cross-layer secure-deletion solutions. Although all solutions affirm the limitations of the legacy storage data path, TrueErase differs in her unique characteristics in using a legacy-compatible, persistent-state-light, centralized, information-propagation channel that runs in parallel with the legacy data path.

The ATA8 TRIM command is implemented on some flash drives to improve performance. It allows the file system to specify blocks that are no longer in-use, and the drive can discard them through internal garbage collection. TRIM was not meant to be a secure-deletion substitute, and it does not guarantee data deletion [SH07]. A current study showed that up to 27% of blocks were recoverable on a TRIM-enabled device [KI11]. Regardless, TRIM does not ensure secure deletion of file metadata.

A semantically-smart-disk system (SDS) [SI03] observes disk requests and deduces common file-system-level information such as block types. The File-Aware Data-Erasing Disk (FADED) is an ext2-based SDS that overwrites deleted files at the file-system layer. Since FADED cannot definitively infer the blocks to be securely deleted (due to reordering), it has to conservatively leave the blocks undeleted at times.

A type-safe disk [SI06] 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. [2008] have modified YAFFS, a log-structured file system for NAND, to handle secure file deletion. The modified YAFFS encrypts files and stores each file's key along with the file's metadata. Whenever a file is deleted, its key is erased, and the encrypted data blocks remain. Sun et al. [2008] modified YAFFS and exploited certain types of NAND flash that allow overwriting of pages to achieve secure deletion.

9. Lessons Learned and Conclusion

This paper presents our third version of TrueErase. Overall, we found retrofitting security features to the legacy storage data path is more complex than we thought. Our first version aimed to bypass many legacy complexities by allowing the file-system layer to securely allocate and deallocate raw storage directly. However, we found that asynchrony and optimizations such as request cancelling and postponing metadata commits via in-memory journal copies made it hard to pinpoint the deallocation times, if they occurred at all.

Our first version was also flash-centric, and we discovered the general lack of raw flash accesses and development environments. Although vendors aim to hide complexities of flash internals without exposing the controls and details for data layout and removal, various internal optimizations (e.g., caching) and reorganizations (e.g., wear leveling) can break file-system-integrity properties and prevent features such as secure deletion.

Our second design used TAP and could work with various file systems and storage media. However, we had a limited understanding of how to apply the theoretical file-system-integrity properties, and were unable to see the structure of corner cases. Also, our TAP implementation was not amenable to state-space enumeration.

Finally, our third iteration took the file-system-integrity properties and verification into consideration, leading to this current incarnation of TrueErase. In retrospect, TrueErase would not be possible without a holistic solution, which highlights the importance of integrating knowledge across often isolated research areas separated by layers and research fields.

To summarize, we have presented the design, implementation, evaluation, and verification of TrueErase,

a legacy-compatible, per-file, secure-deletion framework. We have identified and overcome the challenges of specifying and propagating information across storage layers. We have verified TrueErase and its core logic via cases derived from file-system-integrity properties and state-space enumeration. Although a secure-deletion solution that can withstand diverse threats remains elusive, TrueErase shows a promising step toward this goal.

Acknowledgements

We thank Peter Reiher 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.

References

[BE04] Bennison PF, Lasher PJ. Data security issues relating to end of life equipment. *Proc. of IEEE Int. Symposium on Electronics and the Environment*, 2004.

[BR07] Breeuwsma M, de Jongh M, Klaver C, van der Knijff R, Roeloffs M. Forensic data recovery from flash memory. *Small Scale Digital Device Forensics Journal,* 1(1):1-17, June 2007.

[CO07] Cooke J. Flash memory technology direction. *Proc. of the Windows Hardware Engineering Conf.*, 2007.

[DO95] U.S. DoD. National Industrial Security Program Operating Manual, 5220.22-M, U. S. Government Printing Office, 1995.

[GA01] Ganger GR. Blurring the line between OSes and storage devices. Technical Report CMU-CS-01-166, CMU, 2001.

[GA03] Garfunkel S, Shelat A. Remembrance of data passed: a study of disk sanitization practices, *IEEE Security and Privacy*, 1(1):17-27, January 2003.

[GA10] Garlick J. Scrub utility. computing.llnl.gov, 2010.

[GO11] Goedert, Joseph. Indiana University Health Data Breach Affects 3,000+, www.healthdatamanagement.com, 2011.

[HA08] Halderman JA, Schoen SD, Heninger N, Clarkson W, Paul W, Calandrino JA, Feldman AJ, Appelbaum J, Felten EW. Lest we remember: cold boot attacks on encryption keys, *USENIX Security*, 2008.

[HU02] Hughes G, Wise drives, *IEEE Spectrum,* pp. 37-41, August 2002.

[HU09] Hughes GF, Commins DM, Coughlin T, Disposal of disk and tape data by secure sanitization, *IEEE Security and Privacy*, 7(4):29-34, 2009.

[IR11] Ironkey. www.ironkey.com, 2011.

[JO09] Jones A, Dardick GS, Sutherland I, Valli C. The 2007 analysis of information remaining on disks offered for sale on the second hand market. *Int. Journal of Liability and Scientific Enquiry,* 2(1): 53-68, 2009.

[JO05] Joukov N, Zadok E. Adding secure deletion to your favorite file system. *StorageSS*, 2005.

[JO06] Joukov N, Papaxenopoulos H, Zadok E. Secure deletion myths, issues, and solutions. *StorageSS*, 2006.

[KA97] Katcher J, PostMark: A new file system benchmark, Technical Report TR3022, Network Appliance Inc., 1997.

[KI08] Kingston Technology Data Traveler BlackBox USB Flash Drive Receives FIPS 140-2 Certification. www.kingston.com, 2008.

[KI11] King C, Vidas T, Empirical analysis of solid state disk data retention when used with contemporary operating systems, Digital Investigation, *Proc. of the 11th Annual DFRWS Conf.,* 2011.

[KL86] Kleiman SR, Vnodes: An architecture for multiple file system types in Sun UNIX. *USENIX ATC*, 1986.

[KO07] Koch W. The GNU privacy guard. www.gnupg.org, 2007.

[LE08] Lee J, Heo J, Cho Y, Hong J, Shin SY. Secure deletion for NAND flash file system. *Proc. of the 2008 ACM SAC*, 2008.

[LE11] Lerner M, Kennedy T. Stolen laptop puts thousands at risk of identity theft, http://www.startribune.com/lifestyle/wellness/130644048.html, September 2011.

[NE09] Nester. Wipe. wipe.sourceforge.net, 2009.

[NI06] U.S. NIST. *Special Publication 800-88: Guidelines for Media Sanitization,* September 2006.

[OP11] OpenSSH. www.openssh.com, 2011.

[OS11] OSS-Spectrum Project. Disposition of computer hard drives: specifications for sanitization of hard drives, attachment 2. oss-spectrum.org, 2011.

[PE05] Peterson ZNJ, Burns R, Herring J, Stubblefield A, Rubin AD. Secure deletion for a versioning file system. *USENIX FAST*, 2005.

[PL11] Shred. UNIX man page. htunixhelp.ed.ac.uk, 2011.

[PS11] Children's details lost in laptop theft, www.publicservice.co.uk, 2011.

[SA08] SanDisk Cruzer Enterprise FIPS Edition. www.sandisk.com, 2008.

[SH07] Shu F, Obr N. Data set management commands proposal for ATA8-ACS2, www.t13.org, 2007.

[SI03] Sivathanu M, Prabhakaran V, Popovici FI, Denehy TE, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Semantically-smart disk systems. *USENIX FAST*, 2003.

[SI04] Sivathanu M, Bairavasundaram LN, Arpaci-Dusseau, AC, Arpaci-Dusseau, RH. Life or death at block-level. *OSDI*, 2004.

[SI05] Sivathanu M, Arpaci-Dusseau AC, Arpaci-Dusseau RH, Jha S, A logic of file systems, *USENIX FAST*, 2005.

[SI06] Sivathanu G, Sundararaman S, Zadok E. Type-safe disks. *OSDI*, 2006.

[SM11] Smith J. Mcrypt. mcrypt.sourceforge.net, 2011.

[SU08] Sun K, Choi J, Lee D, Noh SH. Models and design of an adaptive hybrid scheme for secure deletion of data

in consumer electronics. *IEEE Trans. on Consumer Electronics*, 54(1):100-104, February 2008.

[TH06] Thibadeau R, Trusted computing for disk drives and other peripherals, *IEEE Security & Privacy*, 4(5):26-33, 2006.

[VE11a] Store 'n' Go Corporate Secure–FIPS Edition. www.verbatim.com, 2011.

[VE11b] Versel, Neil. Military health plan data breach threatens 4.9 million, www.informationweek.com, 2011.

[WE11] Wei M, Grupp LM, Spada FE, Swanson S. Reliably erasing data from flash-based solid state drives. *USENIX FAST,* 2011.

[YO11] Young E, Hudson T. OpenSSL. www.openssl.org, 2011.