SoK: Secure Data Deletion

Joel Reardon, David Basin, Srdjan Capkun
Institute of Information Security
ETH Zurich
Zurich, Switzerland
{*reardonj,basin,capkuns*}*@inf.ethz.ch*

***Abstract*—Secure data deletion is the task of deleting data irrecoverably from a physical medium. In the digital world, data is not securely deleted by default; instead, many approaches add secure deletion to existing physical medium interfaces. Interfaces to the physical medium exist at different layers, such as user-level applications, the file system, the device driver, etc. Depending on which interface is used, the properties of an approach can differ significantly.**

In this paper, we survey the related work in detail and organize existing approaches in terms of their interfaces to physical media. We further present a taxonomy of adversaries differing in their capabilities as well as a systematization for the characteristics of secure deletion approaches. Characteristics include environmental assumptions, such as how the interface's use affects the physical medium, as well as behavioural properties of the approach such as the deletion latency and physical wear. We perform experiments to test a selection of approaches on a variety of file systems and analyze the assumptions made in practice.

***Keywords*-Secure deletion, Flash memory, Magnetic memory, File systems**

I. Introduction

Secure data deletion is the task of deleting data from a physical medium so that the data is irrecoverable. In the physical world, the importance of secure deletion is well understood: sensitive mail is shredded; published government information is selectively redacted; access to top secret documents is managed to ensure all copies can be destroyed. In the digital world, the importance of secure deletion is also well recognized. Legislative or corporate requirements, particularly relating to privileged or confidential communications, may require secure deletion to avoid the disclosure of sensitive data after physical medium disposal [1], [2]. Regulations may change or new ones enforced causing data assets to become data liabilities, resulting in an immediate need to securely delete a vast amount of data. An example of this is the court ruling that Google's collection of personal wireless network data in Germany was illegal and this data must be destroyed [3]. The importance of secure data deletion is also well-understood by militaries [4].

All modern file systems allow users to "delete" their files. However, they all implement file deletions by *unlinking* files. Abstractly, unlinking a file only changes file system metadata to *indicate* that the file is now "deleted"; the file's full contents remain available. The implicit assumption made is that users delete data not to make the data inaccessible, but simply to recover the consumed storage resources. Despite that, clearing the web browsing history is labelled as a privacy option in most modern web browsers. Mobile phones offer users to delete individual text messages, to clear call logs, etc. In all these systems, users typically assume, falsely, that when they delete the data, it is from that moment on irrecoverable.

There are also many security schemes that require secure deletion to achieve other security goals. A tacit assumption in many key negotiation protocols is that temporary values used to negotiate session keys are not disclosed to an adversary; these must be securely deleted from the system to ensure that the session key is irrecoverable. Several recently proposed systems such as Off-the-Record communication [5], Ephemerizer [6], Vanish [7] and others [8] aim to protect the confidentiality of multi-party communication even in the presence of coercive adversaries, and they explicitly require keys or other protocol parameters to be securely deleted during or after protocol executions.

Secure data deletion has gone by many names, and thus we hear of data being forgotten [6], [9], erased [9]–[12], deleted [11], completely removed [9], [13], reliably removed [12], purged [14]–[16], self-destructed [7], [17], sanitized [12], [14], revoked [9], [13], assuredly deleted [6], [13], [16], securely deleted [18], [19], and destroyed [6], [9], [12]. Whether explicitly stated as a system requirement or implicitly assumed, and however named, the ability to securely delete data in the presence of an appropriate adversary is required for the security of many schemes.

Secure Data Deletion: We say that *data is securely deleted from a system if an adversary that is given some manner of access to the system is not able to recover the deleted data from the system.* This work concerns the secure deletion of *data* that the owner can address by means of a handle, such as a file name or a database record.

To clarify the scope of this systematization, we emphasize that we limit our discussion to the context of secure data deletion, not information deletion. We do not consider deleting all copies of some information, or finding and deleting all derivative works of data. We also do not address deleting all database records containing some sensitive data on a particular topic or all data based on tracking the flow of

information through a complex system. Moreover, we do not consider forensic methods aimed at recovering data objects from scattered pieces; we assume that an adversary capable of recovering all pieces of a data object can thus recover the data object itself. Finally, we do not consider steganographic data storage or deniable encryption. We assume that the data should be made irrecoverable, but we do not worry about leaking the existence of tools to delete the data or encrypted versions of data whose keys are irrecoverable.

Abstractly, the user stores and operates on data objects on a physical medium through an interface. *Data objects* are addressable units of data; these include data blocks, database records, SMS messages, file metadata, entire files, entire hard drives, etc. The *physical medium* is any device capable of storing and retrieving these data objects, such as a magnetic hard drive, a USB stick, or a piece of paper. The *interface* is how the user interacts with the physical medium; the interface offers functions to transform the user's data objects into a form suitable for storage on the physical medium. This transformation can also include operations such as encryption, error-correction, redundancy, etc. In the case of encryption, we assume that computationally-bounded adversaries are only able to recover the original data object if they can also obtain the corresponding decryption key. A coercive adversary may be able to obtain this information while a non-coercive adversary may not. In fact, there are a variety of adversaries one may consider and in Section III we present an adversarial taxonomy.

Related Surveys: Past secure deletion surveys focused on repurposing, where a user disposes a physical medium at a known time and performs a complete sanitization of the medium. Garfinkel et al. [20] survey secure deletion in practice along with the results of the forensic analysis of dozens of hard drives bought on the used market. Diesburg et al. [11] extensively survey confidential data storage methods including the secure deletion of data at its end of life. They also include an analysis of cryptographic methods to store data along with caveats regarding block cipher modes of operation applied to long-term storage. NIST provides a systematization of secure deletion for complete sanitization, detailing precisely which steps must be taken to securely delete data on dozens of media types [14].

However, much of the complexity and nuance of the secure deletion problem is not relevant for repurposing a storage medium, which has a known disclosure time and aims to completely remove all stored data on the physical medium without concern for efficiency, preserving data, or physical wear. In this work we systematize these aspects of secure deletion approaches and show which existing approaches achieve them.

Summary: Our contributions in this paper are the following. We present the many dimensions of secure deletion, consolidating existing notions. We provide a common language to describe secure deletion that unifies the diverse efforts of many researchers and we survey and systematize the related work in detail. We present the approaches in terms of interfaces to physical media: what can be done to achieve secure deletion given a particular interface through which to achieve it? We present a taxonomy of adversaries with different capabilities—adversaries with different manners of access to the storage medium. We explain the relationships among adversaries such that an approach that defeats an adversary also defeats weaker ones; we perform a similar task for characteristics of secure deletion approaches, such as efficiency and granularity. We perform experiments to test a selection of approaches used in practice on a variety of file systems, examining corner cases such as block-unaligned truncations and sparse files. Finally, we analyze which approaches work and what are the best ways to develop a secure deletion solution.

This work is organized as follows. Section II introduces the layers and interfaces involved in accessing a physical medium and surveys secure deletion approaches based on these interfaces. Sections III and IV then present a systematization of the properties of secure deletion adversaries and approaches. Section IV also organizes the approaches from Section II into this framework. Section V compares different approaches and analyzes what works in practice. Finally, Section VI draws conclusions.

II. Secure Deletion by Layers

In this section, we organize secure deletion approaches into the layers through which they access the physical medium. When deciding on a secure deletion approach, one must consider both the interface given to the physical medium and the behaviour of the operations provided by that interface. Secure deletion is typically not implemented by adding a new interface to the physical medium, but rather it is implemented at some existing system layer (e.g., a file system) that offers an interface to the physical medium provided at that layer (e.g., a device driver). It is possible that an interface to a physical medium does not support an implementation of a secure deletion solution.

Once secure deletion is implemented at one layer, then the higher layers' interfaces can explicitly offer this functionality. Care must still be taken to ensure that the secure deletion approach has acceptable performance characteristics: some approaches can be inefficient, cause significant wear, or delete *all* data on the physical medium. These properties are discussed in greater detail in Section IV. For now, we first describe the layers and interfaces involved in accessing magnetic hard drives and flash memory on personal computers, and we explain why there is no one layer that is always the ideal candidate for secure deletion. Afterwards, we present related work in secure deletion organized by the layer in which the approach is integrated.

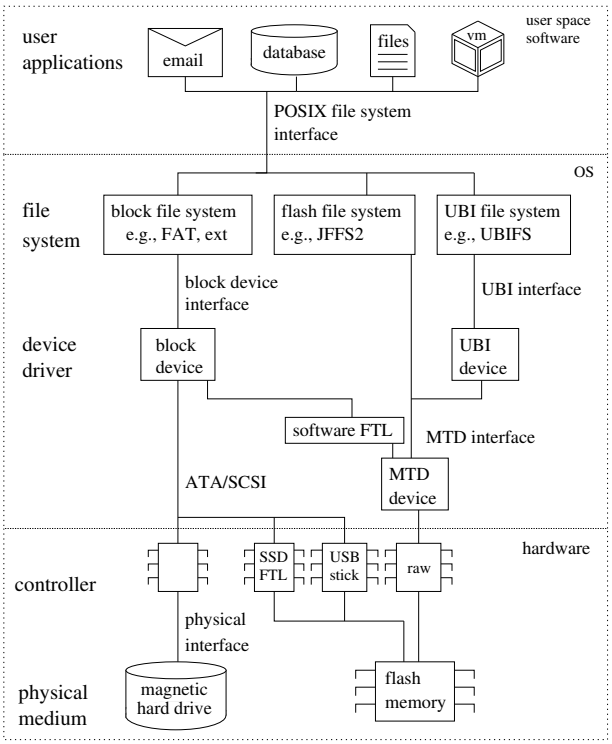

Figure 1. Some interfaces and layers involved in magnetic hard drive and flash memory data storage.

A. Interfaces

Many abstraction layers exist between applications that delete data objects and the physical medium that stores the data objects. While there is no standard sequence of layers that encompass all interfaces to all physical media, Figure 1 shows the typical ways of accessing flash and magnetic storage medium on a personal computer. The lowest layer is always the physical medium itself. Its interface is also physical: depending on the medium it can be degaussed, incinerated, or shredded. Additionally, whatever mechanism controls its operation can be replaced with an ad hoc one; for example, flash memory is often accessed through an obfuscating controller, but the raw memory can still be directly accessed by attaching it to a custom reader [21].

The physical medium is accessed through a controller. The controller is responsible for translating the data format on the physical media (e.g., electrical voltage) into a format suitable for higher layers (e.g., binary values). Controllers offer a standardized, well-defined, hardware interface, such as SCSI or ATA [22], which allow reading and writing to logical fixed-sized blocks on the physical medium. They may also offer a secure erase command that securely deletes *all* data on the physical device [23]. Like physical destruction, this command cannot be used to securely delete some data while retaining other data; we revisit secure deletion *granularity* later in our systematization.

While hard disk controllers consistently map each logical block to some storage location on the physical medium, the behaviour of other controllers differs. Flash memory, notably, behaves differently than magnetic hard drives as it does not permit in-place updates. When raw flash memory is accessed directly, a different controller interface is exposed. For convenience, flash memory is often accessed through a flash translation layer (FTL) controller, whose interface mimics that of a hard drive. FTLs remap logical block addresses to physical locations such that overwriting an old location does not replace it but rather results in two versions, with obvious complications for secure deletion.

Device drivers are software abstractions that consolidate access to different types of hardware by exposing a common simple interface. The block device driver interface allows the reading and writing of logically-addressed blocks. Another device driver—the memory technology device (MTD)—is used to access raw flash memory directly. MTD permits reading and writing, but blocks must be *erased* before being written, and erasing blocks occurs at a large granularity. Unsorted block images (UBI) is another interface for accessing flash memory, which builds upon the MTD interface and simplifies some aspects of using raw flash memory [24]. Figure 1 illustrates all these layers and interfaces.

The device driver interface is used by the file system, which is responsible for organizing logical sequences of data (files) among the available blocks on the physical medium. A file system allows files to be read, written, created, and unlinked. While secure deletion is not a feature of this interface, file systems do keep track of data that is no longer needed. Whenever a file is unlinked, truncated, or overwritten, this is recorded by the file system. The POSIX standard is ubiquitously used as the interface to file systems [25], and the operating system restricts this interface further with access control and permissions.

Finally, the highest layer is user applications. These offer an interface to the user—such as a graphical user interface—that is manipulated by human interface devices such as keyboards and mice. Secure deletion at this layer can be integrated into existing applications, such as an email client with a secure deletion option, or it can be a stand-alone tool that securely deletes all deleted data on the file system.

The choice of layer for a secure deletion approach is a trade off between two factors. At the physical layer, we can ensure that the data is truly irrecoverable. At the user layer, we can easily identify the data object to make irrecoverable. Let us consider these factors in more detail.

Each new abstraction layer impedes direct access to the physical medium, thus complicating secure deletion. The controller may write new data, but the physical medium retains remnants; the file system may overwrite a logical block, but the device driver remaps it physically. The further one's interface is abstracted away from the physical medium, the more difficult it is to ensure that one's actions truly result in the irrecoverability of data.

While low-layer approaches are best suited to ensure irrecoverability, high-layer approaches most easily identify which data objects to delete, e.g., by deleting an email or a file. Indirect information is given to the file system, e.g., by unlinking a file. However, no information is given to the device driver or controller. Assuming the user cannot identify the physical location of the deleted data object on the medium, then an approach integrated at low layers cannot identify where the deleted data object is located. Approaches implemented in the file system tend to strike a suitable balance in this trade off. When this layer is insufficient to achieve secure deletion, it is also possible to pass information on deleted data objects from the file system down to lower layers [18], [26].

In the remainder of this section, we examine secure deletion approaches at the different layers in Figure 1. First, we look at *device-level approaches* and *controller-level approaches*, which have no file system information and therefore securely delete all data on the physical medium. We then move to the other extreme and consider *user-level approaches*, which are easy to install and use, but are limited in their POSIX-level access to the physical medium and are often rendered useless by advanced file system features. Next, we look at *file-system-level approaches* for a variety of systems: (i) those using in-place updates and thus are suitable for magnetic physical media, and (ii) those not using in-place updates and thus are suitable for other physical media including flash memory. We conclude with several approaches that extend existing interfaces to allow information on deleted blocks to be sent to lower layers.

B. Physical-Layer and Controller-Layer Sanitization

Physical Layer: The physical layer's interface is the set of physical actions one can perform on the medium. Secure deletion at this layer often entails physical destruction, but the use of other tools such as degaussers is also feasible. NIST describes the steps required to physically destroy a variety of media [14]. For example, floppy disks must be shredded or incinerated; compact discs must be incinerated or subjected to an optical disk grinding device. Not all approaches work for all media types. For example, most media's physical interfaces permit the media to be put into a NSA/CSS-approved degausser, but this is only a secure deletion approach for particular media types. Magnetic media are securely deleted in this way, while others, such as flash memory, are not.

Controller-Layer: A physical medium is often operated by a controller that translates between the analog medium's analog format and the data format used at higher layers. Several standardized interfaces exist for controllers that permit reading and writing of fixed-sized blocks.

Given these interfaces, there are different actions one can take to securely delete data. Either a single block can be overwritten with a new value to displace the old one, or all blocks can be overwritten. As we noted earlier, with knowledge of neither deleted data nor the organization of data objects into blocks, sanitizing a single block cannot guarantee that any particular data object is securely deleted. Therefore, the controller must sanitize every block to achieve secure deletion. Indeed, both SCSI and ATA offer such a sanitization command, called either *secure erase* or *security initialize* [23]. They work like a button that erases all data on the device by exhaustively overwriting every block. The use of these commands is encouraged by NIST as the non-destructive way to securely delete magnetic hard drives.

An important caveat exists at the physical layer. Controllers translate analog values into binary values such that a range of analog values maps to a single binary value. Gutmann observed that, for magnetic media, the precise analog voltage of a stored bit offers insight into its previously held values [27]. Gutmann's approach to delete this data is also at the controller layer: the controller overwrites every block 35-times with specific patterns. While more recent research was unable to recover overwritten data on modern hard drives [28], it remains safe to say that each additional overwrite does not make the data easier to recover—in the worst case it simply provides no additional benefit. More generally, Gutmann's results highlight that analog remnants introduced by the controller's use of the physical medium may exist for any storage media type and this must be considered when developing secure deletion solutions.

Some flash-based solid-state drives also implement the secure erase feature in their hardware controllers. A study by Wei et al. [21] observed that the implementation of this proceedure is not always correct; in some cases the device reported a successful operation while the entire file system remained available. In follow-up work, Swanson et al. [12] describe an approach for verifiable full-device sanitization that they compare to hard drive degaussing. They propose to encrypt all data written to the physical medium with a key stored only on the hardware controller. To sanitize the device, first the controller's key memory is erased. Every block on the device is then erased, written with a known pattern, and erased again. Finally, the device is reinitialized and a new key given to the flash controller.

C. User-Level Approaches

Device-level approaches interact at the lowest layer and securely delete all data, serving as a useful starting point in our systematization. Now we move to the other extreme, a secure deletion user-level application that can only interact with a POSIX-compliant file system. There are three common user-level approaches: (i) ones that call a secure deletion routine in the physical medium's interface, (ii) ones that overwrite data before unlinking, and (iii) ones that first unlink and then fill the empty capacity of the physical medium.

Secure Erase: UCSD offers a free Secure Erase utility [23]. It is a user-level application that securely erases all data on a storage medium by invoking the Secure Erase command in the hardware controller's interface. This nicely illustrates the propagation of an explicit secure deletion approach to a higher layer interface.

File Overwriting Tools: Another class of user-level secure deletion approaches opens up a file from user-space and *overwrites* its contents with new, insensitive data, e.g., all zeros. When the file is later unlinked, only the contents of the most recent version are stored on the physical medium. To combat analog remnants, overwriting is performed multiple times; multiple tools [29], [30] offer 35-pass overwriting as proposed by Gutmann [27]. Overwriting tools may also attempt to overwrite file metadata, such as the file's name, size, and access times. However, the operating system's interface to the file system may not permit all types of metadata to be arbitrarily changed.

Overwriting tools rely on the following file system property: each file block is stored at known locations and when the file block is updated, then all old versions are replaced with the new version. If this assumption is not satisfied, user-level overwriting tools silently fail. We explore this issue in more detail in Section V.

Free-Space Filling Tools: A file system has valid and unused blocks. The set of unused blocks is a superset of the blocks containing deleted sensitive data, because it is the unused blocks that may still store old, sensitive data. A third class of secure deletion tools exploits this fact by *filling* the entire free space of the file system. This ensures that *all* unused blocks of the physical medium no longer contain sensitive information and instead store new data. These tools also allow secure deletion for file systems that do not perform in-place updates of file data.

The cost of filling is proportional to the free space on the physical medium: the larger the free space, the longer it will take to fill it. In the case of flash memory, where filling the physical medium incurs high wear, the efficiency can be greatly improved by perpetually maintaining the free space of the physical medium within a target range [31]. Examples include Apple's Disk Utility's *erase free space* feature [32] and the open-source tool `scrub` [33]. Filling is also the only user-level means of securely deleting data on an Android phone running YAFFS [31].

Filling tools rely on two assumptions: the user who runs the tool must have sufficient privileges to fill the physical medium to capacity, and when the file system reports itself as unwritable it must no longer contain any deleted data. In Section V, we perform experiments to determine in which file systems the latter condition is satisfied.

Revocable Backup System: Boneh and Lipton propose the first scheme that uses secure deletion of cryptographic keys to securely delete encrypted data under computational assumptions [9]. They created a revocable backup system for off-line (i.e., tape) archives consisting of three user-level applications. Backup files are made revocable *before* writing them to tape. Backups are revoked and then securely deleted without needing physical access to the tapes on which they are stored. Each backup is encrypted with a unique key; each key is then encrypted with a temporary master key. Time is discretized into intervals and each interval is assigned a new master key that encrypts all the backup keys. Backups are periodically deleted from the archive simply by not re-encrypting the corresponding encryption key with the new master key. They extend their user interface to include master key management with a *secure deletion* feature; in their work they propose to write the new key on paper or on a floppy disk and then physically destroy the previous one.

Database Secure Deletion: Databases such as MySQL [34] and SQLite [35] store an entire database as a single file within a file system [36]; databases are analogous to file systems, where records can be added, removed, and updated. This adds a new interface layer for users wanting to delete entries from a database. Database files are long lived on a system, however the data they contain may reside within it very shortly. Many applications store sensitive user data (e.g., emails and text messages) in databases; secure deletion of such data from databases is therefore important.

Both MySQL and SQLite have secure deletion features. In both cases, the interface for secure deletion is the underlying file system and secure deletion is implemented by overwriting the data with zeros. For MySQL, researchers proposed an approach where deleted entries are overwritten with zeros, and the transaction log (used to recover after a crash) is encrypted and securely deleted by deleting the encryption key [36]. For SQLite, developers added a compile-time option to enable a secure deletion feature that overwrites deleted database records with zeros [37].

As previously discussed, overwriting the blocks with zeros is one way to inform the file system that these blocks are unneeded; necessary, but not sufficient, to achieve secure deletion. SQLite's approach relies on the file system below to ensure that overwritten data results in its secure deletion.

D. File-System-Level Approaches with In-Place Updates

We have seen that the utility of user-level approaches is hampered by the lack of direct access to the physical medium. Device-level approaches suffer from being generally unable to distinguish deleted data from valid data given that they lack the semantics of the file system. We now look at secure deletion approaches integrated in the file system itself, that is, approaches that access the physical medium using the device driver interface. We first consider approaches that perform in-place updates of data, and afterwards consider those that do not perform such updates.

An in-place update means that the device driver replaces a location on the physical medium with new content. Not all device drivers offer this in their interface, primarily because

not all physical media support in-place updates. All the approaches in this subsection assume that the device driver is capable of such updates. This assumption is valid for block device drivers, which are used for controlling magnetic hard drives and floppy disks.

Secure Deletion for ext2: The second extended file system ext2 [38] for Linux offers a sensitive attribute for files and directories to indicate that secure deletion should be used when deleting the file. While the actual feature was never implemented by the core developers, researchers provided a patch that implements it [19].

Their patch changed the functionality that marks a block as free. It passes freed blocks to a kernel daemon that maintains a list of blocks that must be sanitized. If the free block corresponds to a sensitive file, then the block is added to the work queue instead of being returned to the file system as an empty block. The work queue is sorted to minimize seek times imposed by random access on magnetic media.

The sanitization daemon runs asynchronously, performing sanitization when the system is idle, allowing the user to perceive immediate file deletion. The actual sanitization method used is configurable, from a simple overwrite to repeated overwrites in accordance with various standards.

Secure Deletion for ext3: The third extended file system ext3 [39] succeeded ext2 as the main Linux file system and extended it with a write journal: all data is first written into a journal before being committed to main storage. This improves consistent state recovery after unexpected losses of power by only needing to inspect the journal's recent changes.

Joukov et al. [40] provide two secure deletion approaches for ext3. Their first approach is a small change that provides secure deletion of data (but not metadata) by overwriting it once, which they call *ext3 basic*. Their second approach, *ext3 comprehensive*, provides secure deletion of file data and metadata by overwriting it using a configurable overwriting scheme, such as the 35-pass Gutmann approach. They both provide secure deletion for all data or just those files whose extended attributes include a sensitive flag.

Secure Deletion via Renaming: Joukov et al. [40] present another secure deletion approach through a file system extension, which can be integrated into many existing file systems [41]. Their extension intercepts file system events relevant for secure deletion: unlinking a file and truncating a file. (They assume overwrites occur in place and are not influenced by a journal or log-structured file system.) For unlinking, which corresponds to regular file deletion, their approach instead moves the file into a special secure deletion directory. For truncation, the resulting truncated file is first copied to a new location and the older, larger file is then moved to the special secure deletion directory. Thus, for truncations, their approach must always process the entire file—not just the truncated component. At regular intervals, a background process runs the user-level tool `shred` [42] on all the files in the secure deletion directory.

Purgefs: Purgefs is another file system extension that conveniently adds efficient secure deletion to any block-based file system [43]. It uses block-based overwriting when blocks are returned to the file system's free list, similar to the approach used for ext2. It supports overwriting file data and metadata for all files or just files marked as sensitive.

Secure Deletion for a Versioning File System: A versioning file system shares file data blocks with many versions of a file; one cannot overwrite the data of a particular block without destroying all versions that share that block. Moreover, user-level approaches such as overwriting the file fail to securely delete data because all file modifications are implemented using a copy-on-write semantics [44]—a copy of the file is made (sharing as many blocks as possible with older versions) with a new version for the block now containing only zeros.

Peterson et al. [45] use a cryptographic approach to optimize secure deletion for versioning file systems. They use an all-or-nothing cryptographic transformation [46] to expand each data block into an encrypted data block along with a small key-sized tag that is required to decrypt the data. If any part of the ciphertext is deleted—either the tag or the message—then the entire message is undecipherable. Each time a block is shared with a new version, a new tag is created and stored for that version. Tags are stored sequentially for each file in a separate area of the file system to simplify sequential access to the file under the assumption that a magnetic-disk drive imposes high seek penalties for random access. A specific version of a file can be quickly deleted by overwriting all of that version's tags. Moreover, *all* versions of a particular data block can easily be securely deleted by overwriting the encrypted data block itself.

E. File-System-Level Approaches without In-Place Updates

While many of our surveyed approaches achieve secure deletion by overwriting data, not all device drivers support these in-place updates. Notably, flash memory cannot perform such overwrites and the MTD device driver [47], which accesses raw flash, does not offer such functionality. Therefore, secure deletion approaches integrated into MTD-based file systems cannot rely on traditional secure deletion approaches such as overwriting.

In-place updates are prohibited because of a particular physical characteristic of flash memory [48]. Flash memory must be *erased* before new data is written. The erasure granularity is typically orders of magnitude larger than the read and write granularities. This means that to securely delete data, other physically colocated data must also be deleted (after being replicated to another physical location) even if it belongs to a different file not marked for deletion.

This asymmetry between the write and erase granularities is not limited to flash memory: it manifests itself in physical media comprised of many write-once read-many

units; units that are unerasable but replaceable. Examples include a library of write-once optical discs or a stack of punched cards. All write-once media are unerasable—NIST says they must be physically destroyed to achieve any form of secure deletion [14]—but first valid colocated data must be replicated onto a new disc or card and then the library updated. Therefore, each erase operation performed on such media destroys one of its constituent storage units.

Similarly, media that can be erased but only with a high asymmetry in granularity also suffer from this problem. For example, a tape archive consists of many magnetic tapes, each storing, say, half a terabyte of data. Each tape must be written end-to-end in one operation; data available for archiving is heuristically bundled onto a tape. Later, to securely delete a single backup on the tape, the entire tape is re-written to a new tape with the backup removed or replaced; the old tape is then erased and reused in the tape archive. This operation incurs cost: tapes have a limited erasure lifetime and tape-drive time is an expensive resource for highly-utilized archives.

Flash memory's erasure granularity asymmetry occurs at a small scale. Flash memory consists of an array of *erase blocks*, each of which is subdivided into *pages*. Erasing a flash erase block prepares its pages to store new data, but all existing data will be removed. Erasure on flash also incurs a cost: each erasure causes a small amount of wear; the long-term consequence is the generation of unusable *bad (erase) blocks*. Any secure deletion approach for such media can use the erasure count as a natural efficiency metric as it encompasses both time complexity and physical wear; in general one favours approaches with few erasures.

In what follows, we present solutions designed for flash memory, but they also work for physical media with this write/erasure asymmetry. (Of course, the difference in scale may obviate some concerns while introducing others.) To avoid flash-specific terminology and confusion between blocks (in block devices) and erase blocks (in MTD devices), we use the term *erase unit* for the unit of erasure and *pages* for the unit of read/write.

Compaction: The naive secure deletion approach for physical media with asymmetry between the write and erase granularities is to immediately compact the erase unit that contains the deleted data: copy the valid colocated data elsewhere and execute the erasure operation. This is a costly operation: copying the data costs time and erasing an erase unit may additionally cause wear on the physical medium. However, there is no other immediate secure deletion approach based on erasures that can do better than one erase unit erasure per deletion. Immediate secure deletion requires a minimum of one erasure. Any improvement to further reduce the number of erase units erased per deletion must batch the deletions and perform intermittent secure deletion.

Batched Compaction: One obvious improvement over the naive approach is to intermittently perform compaction-based secure deletion on all the erase units that have accumulated deleted data since the last secure deletion. This approach is no worse than the naive approach in terms of the time and wear, although the deletion latency—the time the user must wait until data is securely deleted—increases. Each time that deleted data units are colocated on an erase unit, the amortized time and wear cost of secure deletion decreases. Indeed, log-structured file systems already perform a similar technique to recover wasted space, which is usually called garbage collection [49]–[52]. In garbage collection, unlike in our case, compaction is performed only on erase units whose wasted space exceeds a heuristically-computed threshold based on the file system's current need for free space. Garbage collection therefore tries to optimize the number of erasures, but does not consider deletion latency.

Per File Secure Deletion: Lee et al. [53] propose a secure deletion approach for YAFFS [51], which is also generalizable to similar file systems. It performs immediate secure deletion of an entire file at the fixed cost of one erase unit compaction. It reduces the erasure cost of secure deletion by only deleting data at the granularity of a file.

Their approach encrypts each file with a unique key stored in every version of the file's header. The file system is modified to store all versions of a file's header on the same erase unit. Whenever erase units storing headers are full, they are compacted to ensure that file encryption keys are only stored on one erase unit. To delete a file, the erase unit storing the key is compacted for secure deletion, thus deleting all file data under computational assumptions with only one erase unit erasure.

DNEFS: Reardon et al. [15] present a generic file system modification that affords secure deletion for physical media that have high erasure costs. They use encryption as a compression technique to reduce the number of erasures required to securely delete data. They modify the file system by encrypting each data block with a unique key, and colocate the keys in a key storage area. Secure deletion is an intermittent operation that replaces the old key storage area with a new version; keys corresponding to deleted data blocks are not propagated to the new version. Thus, all deleted blocks in the file system are securely deleted, under computational assumptions, with a small number of erasures. DNEFS is implemented for the flash file system UBIFS [50].

Scrubbing: We argued that an immediate approach can only improve the erasure efficiency of the naive approach by not using erasures. Wei et al. [21] propose exactly such an approach for flash memory called *scrubbing*. Scrubbing works by draining the electrical charge from flash memory cells—effectively rewriting the memory to contain only zeros. Erasing a flash memory erase unit is the only way to restore the charge to a cell, but cells can be drained using the write operation. The resulting cells are still unusable for new data, but they are immediately void of the sensitive data. Usefully, colocated data on the erase unit remains available.

A concern with scrubbing is that it officially results in undefined behaviour [48]. Due to the possibility of introducing read errors, flash memory specifications discourage multiple overwrites for some memory and prohibit it for others. The authors examine error rates in practice for different memory types and show that it varies widely; for some devices scrubbing causes frequent errors while for others it causes none.

F. Cross-layer Approaches

Recall that ensuring the irrecoverability of data is easiest at the lowest layers, but there is no information available on the stored location of deleted data objects. There are, however, approaches that pass information on deleted data down through the layers, permitting the use of efficient low-layer secure deletion approaches.

Data objects contained in a file are discarded from a file system in three ways: by unlinking the file, by truncating the file past the block, and by updating the data object's value. The information about data blocks that are discarded when unlinking or truncating files, however, remains only known to the file system. The device-driver layer can only infer the obsolescence of an old block when its logical address is overwritten with a new value. Here we present two approaches by which the file system passes information on discarded blocks to the device driver: TRIM commands [26] and TrueErase [18]. In both cases, the file system informs the device that particular blocks are no longer valid, i.e., needed for the file system. With this information, the device driver can implement its own efficient secure deletion without requiring data blocks to be explicitly overwritten by the file system.

TRIM commands are notifications issued from the file system to the device driver to inform the latter about data blocks that no longer store valid file system data. TRIM commands were not designed for secure deletion but rather as an efficiency optimization for flash-based physical media: without TRIM commands, flash memory eventually suffers a thrashing effect where the device driver must assume that the only deletable block on the physical medium is the one that is being overwritten. Nevertheless, there is no reason that a device driver cannot use information from TRIM commands to perform secure deletion: TRIM commands indicate every time a block is discarded—there are no false negatives. However, it is not possible to restrict TRIM commands only to sensitive blocks, which means that the underlying mechanism that securely deletes the data must be efficient.

Diesburg et al. propose TrueErase [18], which provides similar information as TRIM commands but only for all the blocks belonging to files specifically marked as sensitive. They add a new communication channel between the file system and the device driver that forwards information on blocks deleted from the file system. Device drivers are modified to implement immediate secure deletion when provided a deleted block; the device driver is thus able to implement secure deletion using its lower-layer interface. TrueErase is more efficient than TRIM as it only securely erases a subset of sensitive blocks.

G. Conclusions

This concludes our survey of related work on secure deletion. We saw that physical media can be accessed from a variety of layers and that different layers provide different interfaces for secure deletion. In low-layer approaches, fewer assumption must be made about the interface's behaviour, while in high-layer approaches the user can most clearly mark which data objects to delete. For device-level approaches, we discussed different ways the entire device can be sanitized. User-level secure deletion considers how to securely delete data using a POSIX-compliant file system interface. Secure deletion in the file system must use the device driver's interface for the physical medium. We considered two different kinds of device drivers: those that permit in-place updates and those that do not. For physical media that do not have an erasure operation, physical destruction is the only means to achieve secure deletion.

In the next two sections, we systematize the space of secure deletion approaches. We first review adversarial models and afterwards compare the characteristics of existing approaches.

III. Adversarial Model

Secure deletion approaches must be evaluated with respect to an adversary. The adversary's goal is to recover deleted data objects after being given some access to a physical medium that contained some representation of the data objects. In this section, we present the secure deletion adversaries. We develop our adversarial model by abstracting from real-world situations in which secure deletion is relevant, and identifying the classes of adversarial capabilities characterizing these situations. Table I then presents a variety of real-world adversaries systematized by their capabilities.

A. Classes of Adversarial Capabilities

Attack Surface: The attack surface is the physical medium's interface given to the adversary. If deletion is performed securely, data objects should be irrecoverable to an adversary who has unlimited use of the provided interface. NIST divides the attack surface into two categories: *robust-keyboard attacks* and *laboratory attacks*. Robust-keyboard attacks are software attacks: the adversary acts as a device driver and accesses the storage medium through the controller. Laboratory attacks are hardware attacks: the adversary accesses the physical medium through its physical interface. As we have seen, the physical layer may have analog remnants of past data inaccessible at any other layer. While these two surfaces are widely considered in related work, we emphasize that any interface to the physical medium can be a valid attack surface for the adversary.

Access Time: The access time is the time when the adversary obtains access to the medium. Many secure deletion approaches require performing extraordinary sanitization methods before the adversary is given access to the physical medium. If the access time is unpredictable, the user must rely on secure deletion provided by sanitization methods executed as a matter of routine.

The access time is divided in two categories: user-controlled and adversary-controlled. If the access time is user-controlled, then the user can use the physical medium normally and perform as many sanitization procedures as desired before providing it to the adversary. If the access time is adversary-controlled then we do not permit any extraordinary sanitization methods to be executed: the secure deletion approach must rely on some immediate or intermittent sanitization operation that limits the duration that deleted data remains available.

Number of Accesses: Nearly all secure deletion approaches consider an adversary who accesses a physical medium some time after securely deleting the data. One may also consider an adversary who strikes multiple times—accessing the physical medium before the data is written as well as after it is deleted [15].

We therefore differentiate between single- and multiple-access adversaries. An example single-access adversary corresponds to the scenario when a used device is sold on the market; a multiple-access adversary is someone who, for example, deploys malware on a target machine multiple times because it is discovered and cleaned. One could also imagine multiple accesses escalating in strength at each attack.

Credential Revelation: Encrypting data makes it immediately irrecoverable to an adversary that neither has the encryption key (or user passphrase) nor can decrypt data without the corresponding key. However, there are many situations where the adversary is given this information: a legal subpoena, border crossing, or information taken from the user through duress. In these cases, encrypting data is insufficient to achieve secure deletion.

We partition the credential revelation into non-coercive and coercive adversaries. A non-coercive adversary does not obtain the user's passwords and the credentials that protect the data on the physical medium. A coercive adversary, in contrast, obtains this information. It may also be useful for some to consider a weak-password adversary who can obtain the user's password by guessing, by the device not being in a locked state, or by a cold-boot attack [54]. However, this adversary is unable to obtain secrets such as the user's long-term signing key or the value stored on a two-factor authentication token.

Computational Bound: Many secure deletion approaches rely on encrypting data objects and only storing their encrypted form on the medium. The data is made irrecoverable by securely deleting the decryption key [9], [15], [46], [53]. Thus, encryption is used as a compression technique to reduce secure deletion's cost, as only the small key is securely deleted. The security of such approaches must assume that the adversary is computationally bounded to prevent breaking the cryptographic scheme used.

We distinguish between computationally bounded and unbounded adversaries. There is a wealth of adversarial bounds corresponding to a spectrum of non-equivalent computational hardness problems, so others may benefit from dividing this spectrum further. However, for all the approaches discussed in this paper, it suffices to distinguish between adversaries who can break cryptographic standards such as AES [55] and those who cannot.

B. Summary

Adversaries are defined by their capabilities. Table I presents a subset of the combinatorial space of adversaries that correspond to real-world adversaries. The *name* column gives a name for the adversary, taken from related work when possible; this adversarial name is later used in Table II when describing the adversary a solution defeats. The second through sixth columns correspond to the classes of capabilities defined in this section. The *example* column describes a real-world example where the adversary may be found. For instance, while computationally-unbounded adversaries do not really exist, the consideration of such an adversary may reflect a corporate policy on the export of sensitive data or the consequence of using a broken cryptographic scheme.

Observe that each class of adversarial capabilities is *ordered* based on adversarial strength: lower-layer adversaries get richer data from the physical medium, coercive adversaries get passwords in addition to the physical medium, and an adversary who controls the disclosure time can prevent the user from performing an additional extraordinary secure deletion measure. This yields a partial order on adversaries, where an adversary A is *weaker than or equal to* an adversary B if all of A's capabilities are weaker than or equal to B's capabilities. A is strictly *weaker* than B if all of A's capabilities are weaker than or equal to B's and at least one of A's capabilities is weaker than B's. Consequently, a secure deletion approach that defeats an adversary also defeats all weaker adversaries, under this partial ordering. Finally, as expected, A is *stronger* than B if B is weaker than A.

IV. Analysis of Secure Deletion Approaches

Secure deletion approaches have differing characteristics, which we divide into assumptions on the environment and behavioural properties of the approach. *Environmental assumptions* include the expected behaviour of the system underlying the interface; *behavioural properties* include the deletion latency and the wear on the physical medium. If the environmental assumptions are satisfied then the approach's behavioural properties should hold, the most important of

Adversary's Name	Disclosure	Credentials	Bound	Accesses	Surface	Example
internal repurposing	user	non-coercive	bounded	sing/mult	controller	loan your account
external repurposing	user	non-coercive	bounded	single	physical	sell old hardware
advanced forensic	user	non-coercive	unbounded	single	physical	unfathomable forensic power
border crossing	user	coercive	bounded	sing/mult	physical	perjury to not reveal password
unbounded border crossing	user	coercive	unbounded	sing/mult	physical	corporate policy on encrypted data
malware	adversary	non-coercive	bounded	sing/mult	user-level	malicious application
compromised OS	adversary	non-coercive	bounded	sing/mult	block device	malware in the operating system
bounded coercive	adversary	coercive	bounded	single	physical	legal subpoena
unbounded coercive	adversary	coercive	unbounded	single	physical	legal subpoena and broken crypto
bounded peek-a-boo	adversary	coercive	bounded	multiple	physical	legal subpoena with earlier spying
unbounded peek-a-boo	adversary	coercive	unbounded	multiple	physical	legal subpoena, spying, broken crypto

Table I
TAXONOMY OF SECURE DELETION ADVERSARIES.

which is that secure deletion occurs. No guarantee is provided if the assumptions are violated. It may also be the case that stronger assumptions yields an approach with improved properties.

In this section, we describe standard classes of assumptions and properties. Table II organizes the approaches from Section II into this systematization.

A. Classes of Environmental Assumptions

Adversarial Resistance: An important assumption is on the strength and capabilities of the adversary, as defined in Section III. For instance, an approach may only provide secure deletion for computationally-bounded adversaries; the computational bound is an assumption required for the approach to work. An approach's adversarial resistance is a set of adversaries; adversarial resistance assumes that the approach need not defeat any adversary stronger than an adversary in this set.

System Integration: Our survey organizes secure deletion approaches by the interface through which they access the physical medium. The interface that an approach requires is an environmental assumption, which assumes that this interface exists and is available for use. System integration may also include assumptions on the behaviour of the interface with regards to lower layers (e.g., that overwriting a file securely deletes the file at the block layer). For instance, a user-level approach assumes that the user is capable of installing and running the application, while a file-system-level approach assumes that the user can change the operating system that accesses the physical medium. The ability to integrate approaches at lower-layer interfaces is a stronger assumption than at higher layers because higher-layer interfaces can be simulated at a lower layer.

System integration also makes assumptions about the interface's behaviour. For example, various approaches overwrite data with zeros, assuming that this operation actually replaces all versions of the old data with the new version. As the in-place update assumption is common among approaches, we mark the ones that require it in Table II using the label *"in"* after the integration layer name.

B. Classes of Behavioural Properties

Deletion Granularity: The granularity of an approach is the approach's deletion unit. We divide granularity into three categories: *per-physical-medium*, *per-file*, and *per-data-block*. A per-physical-medium approach deletes *all* data on a physical medium. Consequently, it is an extraordinary measure that is only useful against a *user-controlled access time* adversary, as otherwise the user is required to completely destroy all data as a matter of routine. At the other extreme is sanitizing deleted data at the smallest granularity offered by the physical medium: the data block size (also known as the sector size or page size). Per-data-block approaches securely delete any deleted data from the file system, no matter how small.

Between these extremes lies per-file secure deletion, which targets files as the deletion unit: a file remains available until it is securely deleted. While it is common to reason about secure deletion in the context of files, we caution that the file is not the natural unit of deletion; it often provides similar utility as per-physical-medium deletion. Long-lived files such as databases frequently store user data; the Android phone uses them to store text messages, emails, etc. A virtual machine may store an entire file system within a file: anything deleted from this virtual file system remains until the user deletes the entire virtual machine's storage medium. Consequently, in such settings, per-file secure deletion requires the deletion of all stored data in the DB or VM, which is an extraordinary measure unsuitable against adversaries who control the disclosure time.

Scope: Many secure deletion approaches use the notion of a sensitive file. Instead of securely deleting all deleted data from the file system in an untargeted way, they only securely delete known sensitive files, and require the user to mark sensitive files as such. We divide the approach's scope into *untargeted* and *targeted*. A targeted approach only securely deletes sensitive files, and can substitute for an untargeted approach simply by marking every file as sensitive.

While targeted approaches are more efficient than untargeted ones, we have some reservations about their usefulness. First, the file is not necessarily the correct unit to classify data's sensitivity; an email database is an example of a large file whose content has varying sensitivity. The

benefits of targeting therefore depend on the deployment environment. Second, some approaches do not permit files to be marked as sensitive after their initial creation, such as approaches that must encrypt data objects *before* writing them onto a physical medium. Finally, targeted approaches introduce usability concerns and consequently false classifications due to user error. Users must take deliberate action to mark files as sensitive. A false positive costs efficiency while a false negative may disclose confidential data. While usability can be improved with a tainting-like strategy for sensitivity [56], this is still prone to erroneous labelling and requires user action. Previous work has shown the difficulty of using security software correctly [57] (even the concept of a deleted items folder retaining data confounds some users [58]) and security features that are too hard to use are often circumvented altogether [11].

A useful middle ground is to broadly partition the storage medium into a securely-deleting user-data partition and a normal operating system partition. Untargeted secure deletion is used on the user-data partition to ensure that there are no false negatives and this requires no change in user behaviour or applications. No secure deletion is used for the OS partition to gain efficiency for files trivially identified as insensitive.

Metadata: Deleting a file from a file system deletes the file's data as well as the file's name and other metadata associated with the file. However, most file systems store file metadata separately from the data itself. Consequently, additional steps may be required to securely delete this data when deleting a file. We categorize approaches based on whether they delete metadata as well as data.

Device Lifetime: Some secure deletion approaches incur device wear. We divide device lifetime into complete wear, some wear, and unchanged. *Complete wear* means that the approach physically destroys the medium. *Some wear* means that a non-trivial reduction in the medium's expected lifetime occurs, which may be further subdivided with finer granularity based on notions of wear specific to the physical medium. *Unchanged* means that the secure deletion operation has no significant effect on the physical medium's expected lifetime.

Deletion Latency: Secure deletion latency refers to the timeliness when secure deletion guarantees are provided. There are many ways to measure this, such as how long one expects to wait before deleted data is securely deleted. Here, we divide latency into approaches that offer immediate and delayed secure deletion.

An *immediate* approach is one whose deletion latency is negligibly small. The user is thus assured that data objects are irrecoverable promptly after their deletion. This includes applications that immediately delete data as well as file system approaches that only need to wait until a kernel sanitization thread is scheduled for execution.

A *delayed* approach is one that intermittently executes and provides a larger deletion latency. Such approaches, if run periodically, provide a fixed worst-case upper bound on the deletion latency of all deleted data objects. Delayed approaches involve batching: collecting many pieces of deleted data and securely deleting them simultaneously. This is typically for efficiency reasons [15], [31]. An important factor for delayed approaches is crash-recovery. If data objects are batched for deletion between executions and power is lost, then either the approach must recover all the data to securely delete when restarted (e.g., a commit and replay mechanism like UBIFSec [15]) or it must securely delete all deleted data without requiring persistent state (e.g., filling the hard drive [30], [31], [33], [59]).

Efficiency: Approaches often differ in their efficiency. Wear and deletion latency are two efficiency metrics we explicitly consider. However, other metrics are relevant depending on the application scenario and the physical medium, and so comparing solutions based on these properties may be helpful. Other metrics include the ratio of bytes written to bytes deleted, battery consumption, storage overhead, execution time, etc. The metric chosen depends on the underlying physical medium and use case.

C. Summary

Table II presents the spectrum of secure deletion approaches systematized into the framework developed in this section. For brevity, we do not list all adversaries that an approach can defeat, but instead state what we inferred was the target adversary for which the approach was designed.

The classes of environmental assumptions and behavioural properties are each ordered based on increased *deployment requirements*. Approaches that cause wear and use in-place updates have stronger deployment requirements (i.e., that wear is permitted and the interface allows and correctly implements in-place updates) than approaches that do not cause wear or use in-place updates. Approaches that defeat weak adversaries have stronger deployment requirements (i.e., that the adversary is weak) than approaches that defeat stronger adversaries. The result is a partial ordering on approaches that reflects substitutability: an approach with weaker deployment requirements can replace one with stronger deployment requirements as it requires less to correctly deploy.

V. Case Study: User-Level Approaches

In the previous section, we saw that secure deletion approaches come with assumptions, which, if violated, undermine secure deletion. Understanding the environmental assumptions and determining when they are satisfied is therefore critical to the correct deployment of secure deletion approaches. In this section, we analyze and compare two user-level approaches: overwriting and filling. We describe the assumptions they make on the file-system interface and experimentally determine which file systems satisfy these assumptions.

Solution Name	Target Adversary	Integration	Granularity	Scope	Metadata	Lifetime	Latency	Efficiency
overwrite [29], [30], [42]	unbounded coercive	user-level (in)	per-file	targeted	varies	unchanged	immediate	number of overwrites
fill [32], [33], [59]	unbounded coercive	user-level	per-block	untargeted	varies	unchanged	immediate	depends on medium size
NIST clear [14]	internal repurposing	varies	per-medium	untargeted	varies	varies	immediate	varies with medium type
NIST purge [14]	external repurposing	varies	per-medium	untargeted	varies	varies	immediate	less efficient than clearing
NIST destroy [14]	advanced forensic	physical	per-medium	untargeted	yes	complete wear	immediate	varies with medium type
ATA secure erase [23]	external repurposing	controller	per-medium	untargeted	yes	unchanged	immediate	depends on medium size
flash SAFE [12]	external repurposing	controller	per-medium	untargeted	yes	some wear	immediate	depends on medium size
renaming [40]	unbounded coercive	kernel (in)	per-block	targeted	no	unchanged	immediate	truncations copy the file
ext2 sec del [19]	unbounded coercive	kernel (in)	per-block	targeted	yes	unchanged	immediate	batches to minimize seek
ext3 basic [40]	unbounded coercive	kernel (in)	per-block	targeted	no	unchanged	immediate	batches to minimize seek
ext3 comprehensive [40]	unbounded coercive	kernel (in)	per-block	targeted	yes	unchanged	immediate	slower then ext3 basic
purgefs [43]	unbounded coercive	kernel (in)	per-block	targeted	yes	unchanged	immediate	number of overwrites
ext3cow sec del [45]	bounded coersive	kernel (in)	per-block	untargeted	unknown	unchanged	immediate	deletes multiple versions
compaction	unbounded coercive	kernel	per-block	untargeted	yes	some wear	immediate	inefficient, lots of copying
batched compaction	unbounded coercive	kernel	per-block	untargeted	yes	some wear	delayed	no worse than compaction
per-file encryption [53]	bounded coersive	kernel	per-file	targeted	yes	some wear	immediate	one erasure at file deletion
DNEFS [15]	bounded peek-a-boo	kernel	per-block	untargeted	yes	some wear	delayed	periodic duration trade-off
scrubbing [21]	unbounded coercive	kernel (in)	per-block	untargeted	yes	unchanged	immediate	varies with memory type
ShredDroid [31]	unbounded coercive	user-level	per-block	untargeted	yes	some wear	delayed	depends on medium size

Table II
SPECTRUM OF SECURE DELETION APPROACHES

Overwriting and filling are two general user-level approaches introduced in Section II-C. Overwriting is a targeted per-file approach, while filling is an untargeted per-block approach. Moreover, filling is less efficient than overwriting and is therefore usually run periodically to amortize the deletion cost over multiple data objects. Overwriting requires that every file data block is uniquely stored in the file system, and when it is updated, the old value must be irrecoverable to the adversary, e.g., by overwriting it in place on the physical medium. Filling requires that when the file system claims that it cannot store any more data, then no deleted data remains available on the physical medium. After filling, all the data on the physical medium is used by the current file system.

We performed our experiments using a virtual memory storage medium. We tested block-based file systems using a `loop` device, and MTD-based file systems using `nandsim`. We formatted the device for a particular file system and wrote, overwrote, truncated, and unlinked files with distinct patterns. Afterwards, we unmounted the device and searched for the patterns in memory. We then remounted the device and ran various secure deletion tools, observing the results on the recoverability of the patterns.

File System Data Duplication: Intuitively, overwriting a file's contents makes its previous contents inaccessible. However, many file systems do not replace this data immediately and leave copies of data in multiple locations. An adversary who may access the storage medium at a lower level than the file system can easily recover such copies.

The main reason for data duplication is a file system consistency technique called journalling. Journals colocate the freshest data so that, after an unanticipated power loss, the file system can easily recover to a consistent state by replaying the contents of the journal instead of scanning the entire storage medium. Intermittently, the journal is emptied by committing it to main storage. The journal itself is not securely deleted, so deleted data remains available after committing.

A log-structured file system is a file system that consists only of a journal [60]; there is no notion of main storage. All new data is appended to the journal's end. Consequently, overwriting data writes the new version in an unused location, leaving the old version available until it is later cleaned.

Another class of file systems that do not overwrite old data are copy-on-write file systems. By design, these file systems implement all file writes by copying the old data and updating it at a new location. Both versions remain available, although the old version, if unreferenced, is later deleted to recover the wasted space. Copy-on-write file systems facilitate deduplication, snapshotting, and version control.

Overwriting: We tested secure deletion by overwriting using three different tools: `srm`, `shred`, and `wipe`. The results of these tools are identical and depend only on the file system where the overwritten data is stored. The second

	File Data			File Metadata	
File System	**Updates In-place**	**Filling Works**	**Zeroes Tail**	**Updates In-place**	**Filling Works**
btrfs	no	yes	yes	no	no
exfat	yes	no	no	no	no
ext2	yes	yes	yes	no	no
ext3/ext4	no	yes	yes	no	no
f2fs	no	yes	yes	no	yes
hfs	yes	yes	no	no	no
hfsplus	no	yes	no	no	no
jffs2	no	no	no	no	yes
reiserfs	yes	yes	yes	no	no
ubifs	no	yes	yes	no	yes
vfat	yes	no	no	no	no
yaffs	no	yes	yes	no	yes

Table III
SECURE DELETION AND FILE SYSTEMS

column of Table III shows which file systems overwriting tools manage to securely delete data. As expected, only the simple file systems do this; journalling, log-structured, and copy-on-write file systems all leave old data easily accessible on the file system. Journalling file systems are more amenable to secure deletion by overwriting, however, because committing the journal does indeed securely delete the old version of the data. The original version of the data is still available in the journal, until it is eventually overwritten by new data written to the file system. A user-space overwriting tool can address this by writing sufficient data to overwrite the entire journal. Of course, without the size of the journal (or with a log-structured file system that only consists of a journal), filling the journal effectively requires filling the file system.

Filling: Filling is another user-level approach for secure data deletion that, depending on the file system's implementation, securely deletes all data blocks unreferenced by the file system. This includes truncated and unlinked files, as well as overwritten parts of files for file systems that do not update data in place. The third column of Table III shows the file systems where filling securely deletes data.

Metadata: A file's metadata is data about the file's contents, such as the file's name, access permissions, and creation time. This metadata may also require secure deletion. While overwriting the file's data clearly does not affect its metadata, renaming a file may not actually overwrite the old metadata. Different file systems organize metadata differently; some mix blocks of data and metadata indiscriminately while others store file metadata separately from file data.

The fifth and sixth columns of Table III contain the results of file metadata deletion for overwriting and filling respectively. We used different approaches to securely delete metadata: changing a file using libc's `rename` function [25] (the technique used by `shred`, `srm`, and `wipe`) and unlinking a file and then filling the storage medium. We tested these approaches on different file systems to determine which combinations securely delete metadata. Surprisingly,

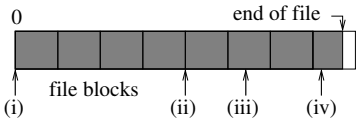

Figure 2. Truncation points in experiments

in all file systems we tested, renaming files rarely removed the old file's name. Even in simple, non-journalling file systems such as FAT and ext2, file metadata is stored in a log-structured manner: an unused directory entry is found to write the new version and then the old version is marked as unused. Naive filling of the file system deletes metadata only for some file systems; however a filling operation that is file-system aware may succeed by creating an appropriate number of files in the appropriate directories to ensure that all metadata storage locations are overwritten.

Truncations: Truncation is an operation on a file that shrinks or extends its size to a specified value. When a truncation extends a file's size, the new data must be initialized to zero. When the file is shrunk, the truncated-over data is discarded. Shrinking truncations are relevant to secure deletion because the discarded data remains available on the storage medium despite it no longer being stored in the file.

We tested truncation behaviour for the following situations: (i) a complete file truncation (size 0), (ii) a block-aligned file truncation, (iii) a block-unaligned file truncation, and (iv) a small file truncation that does not change the file's block count. These situations are illustrated in Figure 2. We then tested different user-level tools' secure deletion of truncated data. For all the file systems we tested, the behaviour of truncation can be expressed with only two components: what the file system does with entirely discarded blocks, and what the file system does with the new last block of the file—called the file's tail—during unaligned truncations.

Our results show that no file system performed any sanitization on entirely discarded blocks. The data they contained remains until the blocks are allocated to a new file. Moreover, overwriting tools no longer delete this data because the size of the file no longer includes this truncated component. Filling tools—if the file system is amenable to filling—securely delete all entirely discarded blocks.

Interestingly, among the file systems we tested, there are two different behaviours after a block-unaligned truncation for the final block. Either the file system rewrites the tail block with zeros from the end of the data to the end of the block, or it leaves it untouched with whatever data was there before truncation. This is listed in the fourth column of Table III. This difference illustrates an eager versus lazy optimization: file systems that rewrite the tail block do not later rewrite them as zeros if the file is truncated to a larger size—it assumes the zero writing has already occurred. In either case, filling solutions are unable to write into tail blocks as the fragmented, unused component is unavailable for allocation. Secure deletion of this data therefore relies

on an interface behavioural assumption: that the file system implements eager overwriting.

Sparse Files: Many file systems offer sparse files: the file's data is assumed to be all zeros unless some data is explicitly stored. In fact, sparse files may have much larger sizes than the physical medium's capacity, which means that overwriting its reported size cannot succeed. None of the overwriting tools treat sparse files in their manuals and all of them overwrite sparse files by naively overwriting their entire reported size. When this size exceeds that of the file system, then they do not securely delete the data once the file system became full: `srm` unlinks the file and exits successfully without output, `shred` fails noisily when it runs out of space, and `wipe` unlinks the file and outputs that the operation was successful, but also reports that the file system became full during the operation.

Currently, once a sparse file part stores data, that part of the file will always explicitly store data. We recommend that future file systems provide hole-punching features for sparse files. This means the file system interprets the explicit writing of zeros blocks as sparse components and does not store the data. Overwriting-based approaches that only write zeros will only unlink the corresponding blocks, leaving the data intact and thus fail to securely delete the data.

Results: Our analysis shows that user-level approaches are generally limited and cannot securely delete data in all cases. However, when users must use such approaches, then a filling-based user-space approach is often preferable to overwriting. It works without complications for a wide range of file systems. Its main drawback is that its run time is proportional to the free space available in the file system. This can be improved, even in user-space, by artificially restricting the free space to a target range [31]. Overwriting, alternatively, requires many assumptions on the file system's behaviour, which do not hold for most file systems. In particular, they only hold for older, simpler file systems, not newer ones with sophisticated features. These unsatisfied assumptions result in many corner-cases where data is not securely deleted. Using the language of our systematization, the environmental assumptions required by overwriting are stronger—and less often satisfied in deployed systems—than those of filling. While overwriting may have preferable behavioural properties (in particular, efficiency), secure deletion and these behavioural properties are only provided if the environmental assumptions are met.

VI. Conclusions

We have explored secure deletion in detail. We defined the simple problem of removing data objects from a physical medium and showed that this problem has many complexities and nuances. We surveyed related work in detail by organizing the approaches in terms of interfaces to the physical medium. We systematized the space of adversaries based on classes of ordered capabilities and related the adversaries to real-world examples; we did the same for the classes of environmental assumptions and behavioural properties. Additionally, we examined two common user-level approaches—showing the limitations of their interfaces by illustrating the complexity of ensuring secure deletion.

Naturally, this survey can be extended. Other storage media types and interfaces exist. Moreover, future developments in physical media may necessitate new kinds of device drivers, while cloud storage may introduce new high-level interfaces. We hope that future work in secure deletion takes advantage of this systematization by also dividing access to the physical medium into layers and implementing secure deletion at the appropriate interface and level of abstraction.

VII. Acknowledgments

This work was partially supported by the Zurich Information Security Center. It represents the views of the authors. We would like to thank our anonymous reviewers as well as Kari Kostiainen, Srdjan Marinovic, Christina Pöpper, Thomas Themel, and Nils Ole Tippenhauer for their many helpful comments.

References

[1] Privacy Commissioner of Canada, "Personal Information Protection and Electronic Documents Act," 2011.

[2] United States Department of Health and Human Services, "HIPAA Security Guidance," 2006.

[3] Electronic Privacy Information Center, "Investigations of Google Street View," 2012. [Online]. Available: https://epic.org/privacy/streetview/

[4] "United States National Industrial Security Program Operating Manual," July 1997. [Online]. Available: http://www.usaid.gov/policy/ads/500/d522022m.pdf

[5] N. Borisov, I. Goldberg, and E. Brewer, "Off-the-record communication, or, why not to use PGP," in *ACM workshop on Privacy in the electronic society*, 2004, pp. 77–84.

[6] R. Perlman, "The Ephemerizer: Making Data Disappear," Tech. Rep., 2005.

[7] R. Geambasu, T. Kohno, A. A. Levy, and H. M. Levy, "Vanish: increasing data privacy with self-destructing data," in *USENIX Security Symposium*, 2009, pp. 299–316.

[8] C. Pöpper, D. Basin, S. Capkun, and C. Cremers, "Keeping Data Secret under Full Compromise using Porter Devices," in *Computer Security Applications Conference*, 2010, pp. 241–250.

[9] D. Boneh and R. J. Lipton, "A Revocable Backup System," in *USENIX Security Symposium*, 1996, pp. 91–96.

[10] N. Provos, "Encrypting virtual memory," in *USENIX Security Symposium*, 2000, pp. 35–44.

[11] S. M. Diesburg and A.-I. A. Wang, "A survey of confidential data storage and deletion methods," *ACM Computing Surveys*, vol. 43, no. 1, pp. 1–37, 2010.

[12] S. Swanson and M. Wei, "SAFE: Fast, Verifiable Sanitization for SSDs," UCSD, Tech. Rep., October 2010.

[13] Y. Tang, P. P. C. Lee, J. C. S. Lui, and R. Perlman, "FADE: Secure Overlay Cloud Storage with File Assured Deletion," in *SecureComm*, 2010, pp. 380–397.

[14] R. Kissel, M. Scholl, S. Skolochenko, and X. Li, "Guidelines for Media Sanitization," September 2006, National Institute of Standards and Technology.

[15] J. Reardon, S. Capkun, and D. Basin, "Data Node Encrypted File System: Efficient Secure Deletion for Flash Memory," in *USENIX Security Symposium*, 2012, pp. 333–348.

[16] A. Rahumed, H. C. H. Chen, Y. Tang, P. P. C. Lee, and J. C. S. Lui, "A Secure Cloud Backup System with Assured Deletion and Version Control," in *ICPP Workshops*, 2011, pp. 160–167.

[17] R. Geambasu, T. Kohno, A. Krishnamurthy, A. Levy, H. Levy, P. Gardner, and V. Moscaritolo, "New directions for self-destructing data systems," University of Washington, Tech. Rep., 2010.

[18] S. Diesburg, C. Meyers, M. Stanovich, M. Mitchell, J. Marshall, J. Gould, A.-I. A. Wang, and G. Kuenning, "TrueErase: Per-File Secure Deletion for the Storage Data Path," *To appear, ACSAC*, 2012.

[19] S. Bauer and N. B. Priyantha, "Secure Data Deletion for Linux File Systems," *Usenix Security Symposium*, pp. 153–164, 2001.

[20] S. Garfinkel and A. Shelat, "Remembrance of Data Passed: A Study of Disk Sanitization Practices," *IEEE Security & Privacy*, pp. 17–27, January 2003.

[21] M. Wei, L. M. Grupp, F. M. Spada, and S. Swanson, "Reliably Erasing Data from Flash-Based Solid State Drives," in *USENIX conference on File and Storage Technologies*, Berkeley, CA, USA, 2011, pp. 105–117.

[22] P. T. McLean, "AT Attachment with Packet Interface Extension (ATA/ATAPI-4)," 1998. [Online]. Available: http://www.t10.org/t13/project/d1153r18-ATA-ATAPI-4.pdf

[23] G. Hughes, T. Coughlin, and D. Commins, "Disposal of disk and tape data by secure sanitization," *Security Privacy, IEEE*, vol. 7, no. 4, pp. 29–34, 2009.

[24] T. Gleixner, F. Haverkamp, and A. Bityutskiy, "UBI - Unsorted Block Images," 2006. [Online]. Available: http://www.linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf

[25] S. Loosemore, R. M. Stallman, R. McGrath, A. Oram, and U. Drepper, "The GNU C Library Reference Manual," 2012. [Online]. Available: http://www.gnu.org/software/libc/manual/pdf/libc.pdf

[26] Intel Corporation, "Intel Solid-State Drive Optimizer," 2009. [Online]. Available: http://download.intel.com/design/flash/nand/mainstream/Intel_SSD_Optimizer_White_Paper.pdf

[27] P. Gutmann, "Secure Deletion of Data from Magnetic and Solid-State Memory," in *USENIX Security Symposium*, 1996, pp. 77–89.

[28] C. Wright, D. Kleiman, and R. S. S. Sundhar, "Overwriting Hard Drive Data: The Great Wiping Controversy," *Information Systems Security*, pp. 243–257, 2008.

[29] D. Jagdmann, "srm - Linux man page."

[30] B. Durak, "wipe - Linux man page."

[31] J. Reardon, C. Marforio, S. Capkun, and D. Basin, "Secure Deletion on Log-structured File Systems," *ASIACCS*, 2012.

[32] Apple, Inc., "Mac OS X: About Disk Utility's erase free space feature," 2012. [Online]. Available: https://support.apple.com/kb/HT3680

[33] J. Garlick, "scrub - linux man page."

[34] Oracle Corporation, "About MySQL," 2012. [Online]. Available: http://www.mysql.com/about/

[35] Hipp, Wyrick & Company, Inc., "About SQLite," 2012. [Online]. Available: http://www.sqlite.org/about.html

[36] P. Stahlberg, G. Miklau, and B. N. Levine, "Threats to privacy in the forensic analysis of database systems," in *ACM SIGMOD conference on Management of data*, 2007, pp. 91–102.

[37] SQLite, "Pragma statements." [Online]. Available: http://www.sqlite.org/pragma.html#pragma_secure_delete

[38] R. Card, T. Ts'o, and S. Tweedie, "Design and Implementation of the Second Extended Filesystem," *Dutch International Symposium on Linux*, 1995. [Online]. Available: http://web.mit.edu/tytso/www/linux/ext2intro.html

[39] S. C. Tweedie, "Journaling the Linux ext2fs Filesystem," *LinuxExpo'98*, 1998.

[40] N. Joukov, H. Papaxenopoulos, and E. Zadok, "Secure Deletion Myths, Issues, and Solutions," *ACM Workshop on Storage Security and Survivability*, pp. 61–66, 2006.

[41] E. Zadok and J. Nieh, "FiST: A Language for Stackable File Systems," in *USENIX Technical Conference*, 2000, pp. 55–70.

[42] C. Plumb, "shred(1) - Linux man page."

[43] N. Joukov and E. Zadokstony, "Adding Secure Deletion to Your Favorite File System," *Third International IEEE Security In Storage Workshop*, pp. 63–70, 2005.

[44] Z. Peterson and R. Burns, "Ext3cow: A Time-Shifting File System for Regulatory Compliance," *Trans. Storage*, vol. 1, no. 2, pp. 190–212, 2005.

[45] Z. Peterson, R. Burns, and J. Herring, "Secure Deletion for a Versioning File System," *USENIX Conference on File and Storage Technologies*, 2005.

[46] R. L. Rivest, "All-Or-Nothing Encryption and The Package Transform," in *Fast Software Encryption Conference*, 1997, pp. 210–218.

[47] "Memory Technology Devices (MTD): Subsystem for Linux," 2008. [Online]. Available: http://www.linux-mtd.infradead.org/

[48] Open NAND Flash Interface, "Open NAND Flash Interface Specification, version 3.0," 2011. [Online]. Available: http://onfi.org/specifications/

[49] D. Woodhouse, "JFFS: The Journalling Flash File System," in *Ottawa Linux Symposium*, 2001. [Online]. Available: http://sources.redhat.com/jffs2/jffs2.pdf

[50] A. Hunter, "A Brief Introduction to the Design of UBIFS," 2008. [Online]. Available: http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf

[51] Charles Manning, "How YAFFS Works," 2010.

[52] J. Kim, "f2fs: introduce flash-friendly file system," 2012. [Online]. Available: https://lkml.org/lkml/2012/10/5/205

[53] J. Lee, S. Yi, J. Heo, and H. Park, "An Efficient Secure Deletion Scheme for Flash File Systems," *Journal of Information Science and Engineering*, pp. 27–38, 2010.

[54] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, pp. 91–98, May 2009.

[55] National Institute of Standards and Technology, "Announcing the Advanced Encryption Standard," 2001.

[56] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "TaintEraser: protecting sensitive data leaks using application-level taint tracking," *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, pp. 142–154, 2011.

[57] A. Whitten and J. D. Tygar, "Why Johnny can't encrypt: a usability evaluation of PGP 5.0," in *USENIX Security Symposium*, 1999, pp. 169–184.

[58] B. Klimt and Y. Yang, "Introducing the Enron Corpus," in *Conference on Email and Anti-Spam*, 2004.

[59] van Hauser, "sfill(1) - Linux man page."

[60] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, vol. 10, pp. 1–15, 1992.