Chapter 2
Related Work on Secure Deletion

2.1 Introduction

This chapter surveys related work and creates a common language of adversaries, behavioural properties, and environmental assumptions by which to compare and contrast related work. In the next chapter, these concepts are used to design the system and adversarial model that we use throughout this book.

The related work presented in this section provides a background to understand the challenges and complications of secure deletion. It is not comprehensive, however, as further related work specific to secure deletion for flash memory and cloud storage are presented in Section 4.3 and 8.3, respectively.

2.2 Related Work

In this section, we organize related work by the layers through which they access the storage medium. When deciding on a secure deletion solution, one must consider both the interface given to the storage medium and the behaviour of the operations provided by that interface. For example, overwriting a file with zeros uses the file system interface, while destroying the medium uses the physical interface. The solutions available to achieve deletion depend on one's interface to the medium. Secure deletion is typically not implemented by adding a new interface to the storage medium, but instead it is implemented at some existing system layer (e.g., a file system) that offers an interface to the storage medium provided at that layer (e.g., a device driver). It is possible that an interface to a storage 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 solution has acceptable performance characteristics: some solutions can be inefficient, cause significant wear, or delete *all* data on the storage medium.

These properties are discussed in greater detail in Section 2.4. For now, we first describe the layers and interfaces involved in accessing magnetic hard drives, flash memory, and network file systems on personal computers. We then 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 solution is integrated.

2.2.1 Layers and Interfaces

Many abstraction layers exist between applications that delete data items and the storage medium that stores the data items. Each of these layers may modify how and where data is actually stored.

While there is no standard sequence of layers that encompass all interfaces to all storage media, Figure 2.1 shows the typical ways of accessing flash, magnetic, and networked storage media on a personal computer.

Physical.

The lowest layer is ultimately physical matter—the actual material that can store data and allow access to data, e.g., paper, tape. 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 [26].

Controller.

The storage medium is accessed through a controller. The controller is responsible for translating the data format on the storage 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 [27], which allows reading and writing to logical fixed-size blocks on the storage medium. They may also offer a *secure erase* command that securely deletes *all* data on the physical device [28]. Like physical destruction, this command cannot be used to securely delete some data while retaining other data; we revisit secure deletion *granularity* later in this chapter.

While hard disk controllers consistently map each logical block to some storage location on the storage medium, the behaviour of other controllers differs. Flash memory does not overwrite existing data; new data is instead logically remapped. When raw flash memory is accessed directly, a different controller interface is exposed. For convenience, flash memory is often accessed through a flash translation
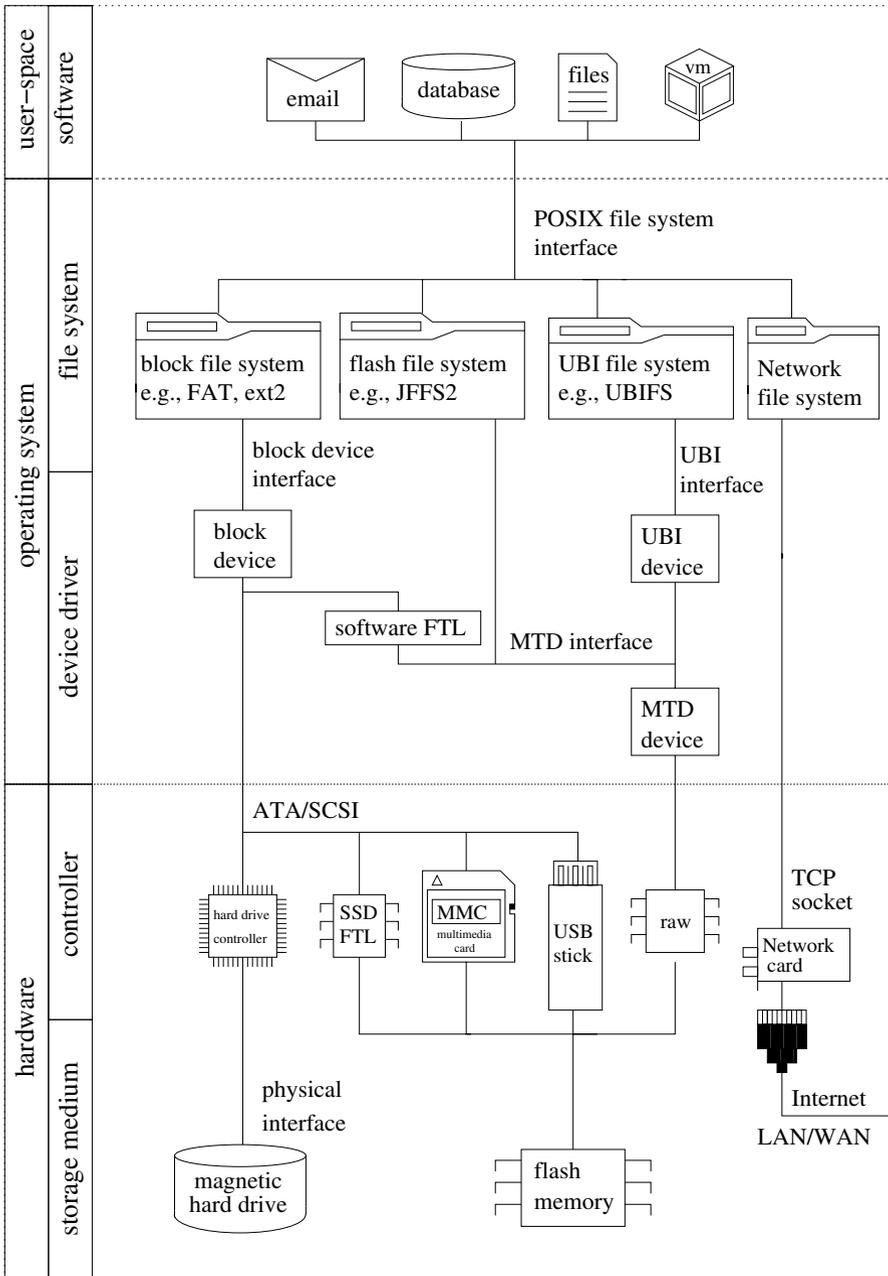
Fig. 2.1 Interfaces and layers involved in magnetic hard drives, flash memory, and remote data storage.

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 instead results in two versions, with obvious complications for secure deletion. FTLs are used in solid-state drives (SSDs), USB sticks, and multimedia cards (MMCs).

Device Driver.

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 [29].

File System.

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 storage 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 [30], and the operating system restricts this interface further with access control.

User Interface.

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

Choosing a Layer.

The choice of layer for a secure-deletion solution 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 item to make irrecoverable. Let us consider these factors in more detail.

Each new abstraction layer impedes direct access to the storage medium, thus complicating secure deletion. The controller may write new data, but the storage 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 storage medium, the more difficult it is to ensure that one's actions truly result in the irrecoverability of data.

High-layer solutions most easily identify which data items 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; no information is given to the device driver or controller. Assuming the user cannot identify the physical location of the deleted data item on the medium, then a solution integrated at low layers cannot identify where the deleted data item is located. Solutions implemented in the file system are usually well balanced in this trade-off. When this layer is insufficient to achieve secure deletion, it is also possible to pass information on deleted data items from the file system down to lower layers [15, 31].

Organization.

In the remainder of this section, we examine existing secure-deletion solutions organized by the different layers in Figure 2.1. First, we look at *device-level solutions* and *controller-level solutions*, which have no file system information and therefore securely delete all data on the storage medium. We then move to the other extreme and consider *user-level solutions*, which are easy to install and use but are limited in their POSIX-level access to the storage medium and are often rendered ineffective by advanced file system features. We then look at a particular set of *file-system-level solutions*: ones that achieve secure data deletion by overwriting the data that is to be deleted. This technique relies on the device driver actually replacing a location on the storage medium with new content—an *in-place* update—and such updates are not possible for all types of storage media. We conclude with several solutions that extend existing interfaces to allow information on deleted blocks to be sent to lower layers. We defer the survey of techniques suitable for flash memory to Chapter 4, and the survey of encryption-based techniques that are suitable for storing data on persistent storage (e.g., remote "cloud" storage) to Chapter 8.

2.2.2 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 provides physical layer secure-deletion solutions suitable for a variety of types of storage media [11]. For example, destroying floppy disks requires shredding or incineration; destroying compact discs requires incineration or subjection to an optical disk grinding device. Of course, not all solutions work for all media types. For example, most media's physical interfaces permit the media to be put into an NSA/CSS-approved degausser, but this is only a secure deletion solution for particular media types. Magnetic media are securely deleted in this way, while others, such as flash memory, are not.

Controller Layer.

Several standardized interfaces exist for controllers that permit reading and writing of fixed-size 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. Because the controller has no knowledge of either the deleted data or the organization of data items into blocks, sanitizing a single block cannot guarantee that any particular data item 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* [28]. 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. The embedded multimedia card (eMMC) specification allows devices to offer a *sanitize* function. If implemented, it must perform the secure deletion of all data (and copies of data) belonging to *unmapped* storage locations, which includes all data that has been previously marked for deletion.

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 [32]. Gutmann's solution to deleting this data is also at the controller layer: the controller overwrites every block 35 times with specific patterns designed to ensure analog remnants are removed for all common data encoding formats. In an epilogue to this work, Gutmann remarks that the 35-pass solution had taken on the mystique of a "voodoo incantation to banish evil spirits," and restates that the reason there are so many passes is that it is the concatenation of the passes required for different data encoding formats; it is *never* necessary to perform all 35 for any specific hard drive.

While more recent research was unable to recover overwritten data on modern hard drives [33], 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 [6]. Gutmann's epilogue states that it is unlikely anything can be recovered from modern drives after a couple of passes with random data. More generally, Gutmann's results highlight that analog remnants introduced by the controller's use of the stor-

age medium may exist for any storage media type and this must be considered when developing secure-deletion solutions.

2.2.3 User-Level Solutions

Device-level solutions 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 securely deleting user-level application that can only interact with a POSIX-compliant file system. There are three common user-level solutions: (i) ones that call a secure deletion routine in the storage medium's interface, (ii) ones that overwrite data before unlinking, and (iii) ones that first unlink and then fill the empty capacity of the storage medium.

Low-Layer Calls.

Device drivers and other low-layer interfaces may expose to user-space special routines for secure deletion. This permits users to easily invoke such functionality without requiring special access to hardware or additional skills. Explicit low-layer calls propagate a secure-deletion solution to a higher-layer interface.

Hughes et al. offer a free Secure Erase utility [28]. 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.

Similarly, Linux's MMC driver exposes to user-space an *ioctl* that invokes the sanitization routine [34]. Therefore, applications can easily call the ioctl, which—if supported by the hardware—performs the secure deletion of all unmapped data on the storage medium.

File Overwriting Tools.

Another class of user-level secure-deletion solutions 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 currently stored on the storage medium. To combat analog remnants, overwriting is performed multiple times; multiple tools [35, 36] offer the 35-pass overwriting as proposed by Gutmann [32].

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. Moreover, they do nothing for larger files that were truncated at some time prior to running the tool. They also do nothing for file copies that are not unlinked with this tool.

Overwriting tools may also attempt to overwrite file system metadata, such as its name, size, and access times. The Linux tool `wipe` [36], for instance, also changes the file name and size in an attempt to securely delete this metadata. Note that not all types of metadata may be arbitrarily changed: the operating system's interface to the file system may not allow it, or simply changing the file's name, for example, may not securely delete the old one. The Linux tool `srm` [35] renames the file to a random value and truncates its size to zero after overwriting. Other attributes cannot be easily changed without higher privileges, e.g., the access times or the file's group and owner.

Overwriting tools operate on either an entire file or the entire storage medium. These tools do not handle operations such as overwrites and truncations, which discard data within a file without deleting the file. Though overwriting a file replaces the old data with the new data (or else such an overwriting tool is unsuitable), it does not perform additional sanitization steps such as writing over the location multiple times. While it is possible to write a user-level tool that securely overwrites and truncates a file as well, it becomes the user's burden to ensure that all other applications make use of it.

This leads into the general problem of usability. The user must remember to use the tool whenever a sensitive file must be deleted, and to do this instead of their routine behaviour. Care must be taken to avoid applications that create and delete their own files [16]: a word processor that creates temporary swap files does not securely delete them with the appropriate tool; a near-exact copy is left available. If a file is copied, the copy too must be securely deleted. Neglecting to use the tool when deleting a file results in the inability to securely delete the file's data with this technique.

Free-Space Filling Tools.

A file system has both valid and unused blocks. The set of unused blocks is a superset of the blocks containing deleted sensitive data. A third class of user-level secure-deletion tools exploits this fact by *filling* the entire free space of the file system. This ensures that *all* unused blocks of the storage medium no longer contain sensitive information and instead store filler material.

Filling solutions permit users and applications to take no special actions to securely delete data; any discarded data is later securely deleted by an intermittent filling operation. These tools also allow secure deletion for file systems that do not perform in-place updates of file data. Compared to the overwriting solutions, secure deletion through filling allows *per-block-level* secure deletion (including truncations) without in-place updates at the cost of a periodic operation. It can only operate at *full scope*—all unused blocks are filled. Examples include Apple's Disk Utility's *erase free space* feature [37] and the open-source tool `scrub` [38].

The correct operation of a filling tool relies on two assumptions: the user who runs the tool must have sufficient privileges to fill the storage medium to capacity, and when the file system reports itself as unwritable it must no longer contain any

deleted data. The useful deployment of these solutions therefore requires manual verification that these assumptions do hold.

Filling's correctness assumptions are satisfied more often than overwriting's correctness assumptions. Modern file systems, for example, typically do not overwrite data in place but instead use journalling when storing new data for crash-recovery purposes. Filling's assumptions, however, also do not always hold. Garfinkel and Malan examine secure deletion by filling for a variety of file systems and find mixed results [39]. One observation is that creating many small files helps securely delete data that is not deleted when creating one big file, which may be due to file systems not allocating heavily fragmented areas for already large files.

The benefits of filling over overwriting are that the user is given secure deletion for all deleted data (including unmarked sensitive files and truncations) that works correctly for a larger set of file systems. Moreover, the user only needs to run the tool periodically to securely delete all accumulated deleted data: applications and user behaviour do not need to change with regards to file management. The trade-off is that the filling operation is slow and cannot target specific files. It is a periodic operation that securely deletes all data blocks discarded within the last period. Since deletion is achieved by writing new data instead of overwriting the existing data, it does not perform in-place updates and is therefore suitable for additional file systems and storage medium types that do not permit such operations.

Database Secure Deletion.

Databases such as MySQL [40] and SQLite [41] store an entire database as a single file within a file system [42]; 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; the data they contain, however, may only be valid for a short time. 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 with in-place updates. For MySQL, researchers propose a solution 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 [42]. For SQLite, there is an optional secure-deletion feature that overwrites deleted database records with zeros [43].

As previously discussed, overwriting 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 solution relies on the file system "below" to ensure that overwritten data results in its secure deletion. When the interface does not explicitly offer secure deletion, then it is—at the minimum—necessary to tell the interface that the data is discarded.

2.2.4 File-System-Level Solutions with In-Place Updates

The utility of user-level solutions is hampered by the lack of direct access to the storage medium. Device-level solutions 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 solutions integrated in the file system itself, that is, solutions that access the storage medium using the device driver interface.

Here we consider only solutions that use in-place updates to achieve secure deletion. An in-place update means that the device driver replaces a location on the storage medium with new content. Not all device drivers offer this in their interface, primarily because not all storage media support in-place updates. The assumption that in-place updates occur is valid for block device drivers that access magnetic hard drives and floppy disks. Solutions for flash memory cannot use in-place updates, and Section 4.3 discusses this in more detail.

Secure Deletion for ext2.

The second extended file system ext2 [44] 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 [16].

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 spinning-disk 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 [45] 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. [46] provide two secure-deletion solutions for ext3. Their first solution is a small change that provides secure deletion of file data by overwriting it once, which they call *ext3 basic*. Their second solution, *ext3 comprehensive*, provides secure deletion of file data and file metadata by overwriting it using a configurable overwriting scheme, such as the 35-pass Gutmann solution. They both pro-

vide secure deletion for all data or just those files whose extended attributes include a sensitive flag.

Secure Deletion via Renaming.

Joukov et al. [46] present another secure-deletion solution through a file system extension, which can be integrated into many existing file systems [47]. 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 solution 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 solution must always process the entire file—not just the truncated component. At regular intervals, a background process runs the user-level tool `shred` [48] on all the files in the secure-deletion directory.

Purgefs.

Purgefs is another file system extension that adds secure deletion to any block-based file system [49]. It uses block-based overwriting when blocks are returned to the file system's free list, similar to the solution used for ext2 [44]. It supports overwriting file data and file metadata for all files or just files marked as sensitive. Purgefs is implemented as a generic file system extension, which can be combined with any block-based file system to create a new file system that offers secure deletion.

Secure Deletion for a Versioning File System.

A versioning file system shares file data blocks among 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 solutions such as overwriting the file fail to securely delete data because all file modifications are implemented using a copy-on-write semantics [50]—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. [51] use a cryptographic solution to optimize secure deletion for versioning file systems. They use an all-or-nothing cryptographic transformation [52] 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.

2.2.5 Cross-layer Solutions

There are solutions that pass information on discarded data down through the layers, permitting the use of efficient low-layer secure-deletion solutions.

Data items 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 item's value. The information about data blocks that are discarded when unlinking or truncating files, however, remains known only by 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 solutions by which the file system passes information on discarded blocks to the device driver: TRIM commands [31] and TrueErase [15]. In both cases, the file system informs the device that particular blocks are discarded, i.e., no longer 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 discarded data blocks. TRIM commands were not designed for secure deletion but instead as an efficiency optimization for flash-based storage media. 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. It is not possible to restrict TRIM commands only to sensitive blocks, which means that it must be an efficient underlying mechanism that securely deletes the data.

Diesburg et al. propose TrueErase [15], which provides similar information as TRIM commands but only for blocks belonging to files specifically marked as sensitive. Users may simply set all files to sensitive or use traditional permission semantics to manage file sensitivity. TrueErase adds a new communication channel between the file system and the device driver that forwards from the former to the latter information on sensitive 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 correctly implement secure deletion using its lower-layer interface with the high-layer information on what needs to be deleted. This is more efficient than TRIM commands, which would require deletion for all data. This comes at the risk of a false negative in the event that a user neglects to correctly set a file's sensitivity.

2.2.6 Summary

This concludes our survey of selected related work on secure deletion. Further related work specific to flash memory and persistent memory appear in Chapters 4 and 8, respectively. We saw that storage media can be accessed from a variety of layers and that different layers provide different interfaces for secure deletion. In low-layer solutions, fewer assumptions must be made about the interface's behaviour, while in high-layer solutions the user can most clearly mark which data items to delete. For device-level solutions, 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 storage medium, and we surveyed solutions that assume the device driver performs in-place updates. For storage media that do not have an erasure operation, physical destruction is the only means to achieve secure deletion.

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

2.3 Adversarial Model

Secure-deletion solutions must be evaluated with respect to an adversary. The adversary's goal is to recover deleted data items after being given some access to a storage medium that contained some representation of the data items. 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 2.1 then presents a variety of real-world adversaries organized by their capabilities.

2.3.1 Classes of Adversarial Capabilities

Attack Surface.

The attack surface is the storage medium's interface given to the adversary. If deletion is performed securely, data items 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* [11]. 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 storage 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 storage 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 solutions require performing extraordinary sanitization methods before the adversary is given access to the storage 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 into two categories: predictable (or user controlled) and unpredictable (or adversary controlled). If the access time is predictable, then the user can use the storage medium normally and perform as many sanitization procedures as desired before providing it to the adversary. If the access time is unpredictable then we do not permit any extraordinary sanitization methods to be executed: the secure-deletion solution 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 solutions consider an adversary who accesses a storage medium some time after securely deleting the data. One may also consider an adversary who accesses the storage medium multiple times—accessing the storage medium before the data is written as well as after it is deleted, similar to the *evil maid* attack on encrypted file systems.

We therefore differentiate between single- and multiple-access adversaries. A single-access adversary may surface when a used storage medium 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, or someone who obtains surreptitious periodic access (e.g., nightly access) to a storage medium.

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. There are many situations, however, 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 storage medium. A coercive adversary, in contrast, obtains this information. It may also be useful 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 [53]. This adversary, however, 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 solutions rely on encrypting data items and only storing their encrypted form on the medium. The data is made irrecoverable by securely deleting the decryption key. The security of such solutions must assume that the adversary is computationally bounded to prevent breaking the cryptography.

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.

2.3.2 Summary

Adversaries are defined by their capabilities. Table 2.1 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 2.3 (and much later in Table 12.1) when describing the adversary a solution defeats. The second through fifth columns correspond to the classes of capabilities defined in this section. Table 2.2 provides a real-world example where each of the adversaries from Table 2.1 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 a risk analysis of a potentially broken cryptographic scheme.

Observe that each class of adversarial capabilities is *ordered* based on adversarial strength: lower-layer adversaries get richer data from the storage medium, coercive adversaries get passwords in addition to the storage medium, and an adversary who controls the disclosure time can prevent the user from performing an additional extraordinary effort to achieve secure deletion. 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 solution that defeats

Table 2.1 Taxonomy of secure-deletion adversaries.

Adversary's Name	Disclosure	Credentials	Bound	Accesses	Surface
internal repurposing	predictable	non-coercive	bounded	sing/mult	controller
external repurposing	predictable	non-coercive	bounded	single	physical
advanced forensic	predictable	non-coercive	unbounded	single	physical
border crossing	predictable	coercive	bounded	sing/mult	physical
unbounded border	predictable	coercive	unbounded	sing/mult	physical
malware	unpredict.	non-coercive	bounded	sing/mult	user-level
compromised OS	unpredict.	either	bounded	sing/mult	block dev
bounded coercive	unpredict.	coercive	bounded	single	physical
unbounded coercive	unpredict.	coercive	unbounded	single	physical
bounded multi-access	unpredict.	coercive	bounded	multiple	physical
unbounded multi-access	unpredict.	coercive	unbounded	multiple	physical

Table 2.2 Example of adversaries modelled.

Adversary's Name	Example
internal repurposing	loaning hardware
external repurposing	selling old hardware
advanced forensic	unfathomable forensic power
border crossing	perjury to not reveal password
unbounded border crossing	cautious corporate policy on encrypted data
malware	malicious application
compromised OS	operating system malware, passwords perhaps provided
bounded coercive	legal subpoena
unbounded coercive	legal subpoena and broken crypto
bounded multiple-access	legal subpoena with earlier spying
unbounded multiple-access	legal subpoena, spying, broken crypto

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*.

2.4 Analysis of Solutions

Secure-deletion solutions have differing characteristics, which we divide into assumptions on the environment and behavioural properties of the solution. *Environmental assumptions* include the expected behaviour of the system underlying the interface; *behavioural properties* include the deletion latency and the wear on the storage medium. If the environmental assumptions are satisfied then the solution's behavioural properties should hold, the most important of which is that secure deletion occurs. No guarantee is provided if the assumptions are violated. It may also be the case that stronger assumptions yield solutions with improved properties.

In this section, we describe standard classes of assumptions and properties. Table 2.3 organizes the solutions from Section 2.2 into this systematization.

2.4.1 Classes of Environmental Assumptions

Adversarial Resistance.

An important assumption is the one made on the strength and capabilities of the adversary, as defined in Section 2.3. For instance, a solution may only provide secure deletion for computationally bounded adversaries; the computational bound is an assumption required for the solution to work. A solution's adversarial resistance is a set of adversaries; adversarial resistance assumes that the solution need not defeat any adversary stronger than an adversary in this set.

System Integration.

This chapter organizes the secure-deletion solutions by the interface through which they access the storage medium. The interface that a solution 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 solution assumes that the user is capable of installing and running the application, while a file-system-level solution assumes that the user can change the operating system that accesses the storage medium. The ability to integrate solutions 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, solutions that overwrite data with zeros assume that this operation actually replaces all versions of the old data with the new version. When such interface assumptions are not satisfied, then the solution does not provide secure deletion. Table 2.3 notes the solutions that require in-place updates in order to correctly function.

2.4.2 Classes of Behavioural Properties

Deletion Granularity.

The granularity of a solution is the solution's deletion unit. We divide granularity into three categories: *per storage medium*, *per file*, and *per data item*. A per-storage-medium solution deletes *all* data on a storage 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 storage medium: e.g., block size, sector size, or page size. Per-data-item solutions 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-storage-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. 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. In other settings, such as the storage of large media files, file data tends to be stored and deleted at the granularity of an entire file and so per-file solutions may reduce overhead.

Scope.

Many secure-deletion solutions 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 solution's scope into *untargeted* and *targeted*. A targeted solution only securely deletes sensitive files and can substitute for an untargeted solution simply by marking every file as sensitive.

While targeted solutions 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 solutions do not permit files to be marked as sensitive after their initial creation, such as solutions that must encrypt data items *before* writing them onto a storage medium. Such solutions are not suitable for cases where, for example, users manually mark emails from the inbox as sensitive so that additional secure-deletion actions are taken when it is later deleted. Finally, targeted solutions 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 [54], this is still prone to erroneous labelling and requires user action. Previous work has shown the difficulty of using security software correctly [55] (even the concept of a deleted

items folder retaining data confounds some users [56]) and security features that are too hard to use are often circumvented altogether [6].

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 deemed not sensitive.

Device Lifetime.

Some secure-deletion solutions incur device wear. We divide device lifetime into complete wear, some wear, and unchanged. *Complete wear* means that the solution 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 storage medium. *Unchanged* means that the secure-deletion operation has no significant effect on the storage 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 immediate and periodic secure deletion.

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

A *periodic* solution is one that intermittently executes and provides a larger deletion latency. Such solutions, if run periodically, provide a fixed worst-case upper bound on the deletion latency of all deleted data items. Periodic solutions involve batching: collecting many pieces of deleted data and securely deleting them simultaneously. This is typically for efficiency reasons. An important factor for periodic solutions is crash-recovery. If data items are batched for deletion between executions and power is lost, then either the solution must recover all the data to securely delete when restarted (e.g., using a commit and replay mechanism) or it must securely delete all deleted data without requiring persistent state (e.g., filling the hard drive [36, 38, 57]).

Table 2.3 Spectrum of secure-deletion solutions

Solution Name	Target Adversary	Integration	Granularity	Scope
overwrite [35, 36, 48]	unbounded coercive	user-level[a]	per-file	targeted
fill [37, 38, 57]	unbounded coercive	user-level	per-data-item	untargeted
NIST clear [11]	internal repurposing	varies	per-medium	untargeted
NIST purge [11]	external repurposing	varies	per-medium	untargeted
NIST destroy [11]	advanced forensic	physical	per-medium	untargeted
ATA secure erase [28]	external repurposing	controller	per-medium	untargeted
renaming [46]	unbounded coercive	kernel[a]	per-data-item	targeted
ext2 sec del [16]	unbounded coercive	kernel[a]	per-data-item	targeted
ext3 basic [46]	unbounded coercive	kernel[a]	per-data-item	targeted
ext3 comprehensive [46]	unbounded coercive	kernel[a]	per-data-item	targeted
purgefs [49]	unbounded coercive	kernel[a]	per-data-item	targeted
ext3cow sec del [51]	bounded coercive	kernel[a]	per-data-item	untargeted

[a] Assumes interface performs in-place updates.

Solution Name	Lifetime	Latency	Efficiency
overwrite [35, 36, 48]	unchanged	immediate	number of overwrites
fill [37, 38, 57]	unchanged	immediate	depends on medium size
NIST clear [11]	varies	immediate	varies with medium type
NIST purge [11]	varies	immediate	less efficient than clearing
NIST destroy [11]	destroyed	immediate	varies with medium type
ATA secure erase [28]	unchanged	immediate	depends on medium size
renaming [46]	unchanged	immediate	truncations copy the file
ext2 sec del [16]	unchanged	immediate	batches to minimize seek
ext3 basic [46]	unchanged	immediate	batches to minimize seek
ext3 comprehensive [46]	unchanged	immediate	slower then ext3 basic
purgefs [49]	unchanged	immediate	number of overwrites
ext3cow sec del [51]	unchanged	immediate	deletes multiple versions

Efficiency.

Solutions often differ in their efficiency. Wear and deletion latency are two efficiency metrics we explicitly consider. The particular relevant metrics depend on the application scenario and the storage medium. 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 storage medium and use case.

2.4.3 Summary

Table 2.3 presents the spectrum of secure-deletion solutions organized into the framework developed in this section. For brevity, we do not list all adversaries that a solution can defeat, but instead state what we inferred as the solution's target adversary.

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

Springer