Secure Data Deletion for the Ext2 File System

Steve Bauer, Bodhi Priyantha

Laboratory for Computer Science

MIT

December 7, 2000

Abstract

Security conscious users of file systems require that deleted information and its associated metadata is no longer accessible on the underlying physical disk. Existing file system implementations only reset the file system data structures to reflect the removal of data, leaving both the actual data and its associated metadata on the physical disk. Even when this information has been overwritten it may remain visible to advanced probing techniques such as magnetic force microscopy or magnetic force scanning tunneling microscopy. Our project addresses this problem by adding support to the *Linux ext2* filesystem for asynchronous secure deletion of file data and metadata using the best known overwriting techniques. An asynchronous overwriting process sacrifices immediate security but ultimately provides a far more usable and complete secure deletion facility. We justify our design by arguing that user level secure deletion tools are inadequate for numerous reasons and that synchronous deletion facilities are too time consuming to be acceptable to users. Further we contend that encrypting file information on disk is not a sufficient solution to achieve security.

1 Introduction

Secure deletion of data has been considered for years and different implementations of secure deletion facilities abound. However from our survey of secure deletion techniques no one yet has implemented it properly and in a usable fashion for any common existing operating system. After we briefly explain our motivation for considering the problem, we will detail the shortcomings of existing techniques for securely deleting information from disks.

Initially our motivation stemmed from considering distributed access to sensitive data and distributed file systems. Users increasingly access their data from many remote locations including home and office machines, terminals in airports and Internet cafes, and multiple workstations in a university's computer clusters. It is possible that these remote machines cache sensitive user data on the local disks. Distributed file systems such as AFS cache user data on the client machines drives.[1] Even if the user data is accessed by a method that does not rely upon local disk caches the user themselves or the applications they use might copy sensitive files to a temporary local directory. For instance popular web browsers cache accessed information on a local drive and clearing the cache does not really remove the data from the underlying disk. Thus users need assurances that sensitive data that they access does not remain visible on every machine they use.

Any implementations of secure deletion at the user level will be inadequate. Numerous user level secure deletion tools already exist. [2, 13, 14, 15] While all of these do overwrite files, only one [13] uses the overwriting patterns suggested by best current security practices. Further all user space programs face four fundamental problems that cannot be addressed effectively.

1. **Secure deletion tools cannot be interposed between all user file operations:** Although it is possible to replace obvious file removal programs such as `rm` with a user level secure deletion tool, this does not work with other less obvious ways of deleting file data, such as replacing a file with the contents of another `cp file existingfile`. To be used with a user level secure deletion mechanism, these commands would have to be wrapped with scripts that would first securely overwrite the existing file before proceeding with the normal file operation. Further a user level deletion tool would have to be integrated with every application that handles handles its own file management.

2. **File truncation cannot be handled effectively**: User level deletion programs only overwrite an entire file and thus cannot handle file truncation. If someone uses an editor to delete half of a file causing some blocks to be returned to the free list that information will not be securely overwritten and will remain visible on the disk. One might imagine handing off a file to a secure deletion tool and telling it to overwrite past a particular offset, however this interferes with any correctness notions an application might have about file contents.

3. **File metadata cannot be overwritten completely:** The sensitive information in a file system includes the contents of a file, and the file metadata including the name, size, owner information, and creation, modification, access, and deletion times. User level programs usually overwrite only the file data itself. Although parts of the metadata could be overwritten by means such as using `touch` to set the file access, creation, and modification times and renaming the file many times to obfuscate the file name, such techniques tend to be cumbersome and time consuming. Even with these workarounds, important file metadata information such as what blocks the file contained, the user and group ownerships, and deletion time would not be removed.

4. **Overwriting synchronously is inconvenient from a user perspective:** When a synchronous deletion tool is used, users are generally unwilling to wait for the overwriting process to complete. While the deletion process could be placed in the background allowing activities to proceed, application correctness may depend upon the file not being in the name space once the deletion tool was called. The deletion tool could rename a file before the overwrite process begins but, renaming of the file is not possible if the tool is to be used for file truncation as well.

Another approach to prevent user data from remaining visible on the disk would be to use encryption. The possible approaches would be to either use an encrypting file system or to use encryption tools to selectively encrypt files on a disk. Either of these is inadequate due to following reasons.

1. **Encryption keys can be revoked or compromised**: Anytime that a key is revoked the data associated with it must be considered accessible. Therefore owners need a mechanism for securely deleting all the files encrypted with a revoked key. Further if a user is compelled by law to turn over their encryption keys the deleted files that remain on the disk are easily accessible.

2. **Encryption inefficiency makes such a solution potentially unusable**: The performance degradation of an encrypting file system compared to a regular file system may be very high. Similarly encryption tools may be too much of a performance drag for a user since the encryption tool must be used for every file operation, not just at the file deletion time. This is the traditional conflict between security and usability.

3. **Plaintext files may remain visible**: Before files are encrypted they may have existed as plaintext stored either in temporary or regular

files. These plain text files must be deleted securely. An encrypting filesystem does not suffer from this problem since all files stored on the filesystem are encrypted.

To address above problems, we have implemented asynchronous secure deletion for the *Linux ext2* file system. Section 2 details the design goals of our system. Section 3 briefly covers some background information explaining how data is stored on disk, how it can be recovered even after it has been overwritten a limited number of times, and the best common practices of overwriting data to prevent recovery.

Section 4 details our implementation. Section 5 explains how we recommend the system be used in practice and evaluation of system performance. Finally Section 6 concludes the paper with a consideration of planned future work including managing the swap disk space and our plans to release the system to the Linux community.

2 Design

Our design focused on addressing two main goals: completeness and transparency. Completeness entails that all the data over the entire course of a files existence must have been securely overwritten. File data read into application buffers and swapped out to a swap disk must be properly deleted as well. We must also properly maintain the security guarantees in the case of a system crash before the disk blocks have been fully overwritten. Similarly if a crash occurs between writing the sensitive data to a disk block and modification of the inode, we need to ensure that the data is properly erased. Thus tools that reconstruct the file system after a crash need to be modified to handle this problem. After these are implemented no remnants of a deleted file should remain anywhere on the physical disks.

Another important component of our completeness requirement is the employment of overwrite patterns that most effectively guarantee that the data will no longer be accessible even to advanced probing techniques. After having reviewed Department of Defense directives for managing data, academic literature, and open source secure deletion tools we concluded that Peter Gutmann's work at the University of Auckland [5] presents the best current security practices for overwriting data. We briefly describe both the overwrite patterns and their justification in the background section.

Our completeness requirement also entails that we consider different aspects of a file system usage such as the disk drive behaviors, system administration policies, and user practices. SCSI disks for instance keep a surprising large on disk cache. The proper mode bits must be set on SCSI drive writes to ensure that data actually is written to the physical disk, Drives that re-map blocks to other sectors upon sector failure must also be considered. Another issue that users must consider when deleting data from a disk is the possibility of having multiple copies of the file, either through their own actions or due to backup policies of the system administrator. While our system cannot address this facet of completeness, we do our best in our system documentation to ensure that the user is aware of all the issues.

Our second goal is to achieve transparency from both a user perspective and system perspective. File deletion and truncation must be very fast to satisfy user expectations. A regular remove operation or file truncation operation can proceed at in memory speeds. When a user deletes half a document that was previously saved, she does not consider the effects of underlying disk blocks being returned to a free list. Similarly when replacing one file with another a user does not typically consider the removal of data blocks from the replaced file that actually occurred within the file system. Users thus are accustomed to data deletion operations being fast and transparent.

Secure removal though is inherently a slow operation that involves ensuring that the underlying disk blocks have been overwritten with data multiple times. Overwriting a file with even two passes of random data involves writing the random data, flushing the data to disk, waiting for the head seek and writing to complete, and then repeating the process all over again. This will take tens of milliseconds for even a small file. Were a user to delete a file with megabytes of data, the removal process would be intolerable slow. A user should be able to delete or truncate a file and proceed immediately with other normal operations. To this end, we use an asynchronous deletion process which takes the deleted blocks and writes over them numerous times. As far as a user is concerned, the user time spent in deleting a file securely is comparable to the time taken for deleting a conventional file.

To ensure transparency disk quotas must also be maintained properly. during the period of time while the blocks are being overwritten. We require that a users disk quota reflect the fact that their disk blocks being overwritten are not available for reassignment. Therefore blocks remain a part of her disk quota until the overwrite process is complete and the blocks are returned to the free list. Without this requirement a user would be able to quickly allocate and securely delete large files, starving other users of disk resources.

3 Background

Recovering deleted data from disk drives is remarkably easy. Most users are not aware that after they delete a file it still remains visible on the disk. Various user space tools that recovery deleted files exist both in the free Unix world and for Windows and Macintosh machines. Prominent ones for Linux are the various file system editors such as the ext2 utilities, and The Corners Toolkit. The ability to recover supposedly deleted files is beneficial to the user who inadvertently removes an important file. However more people would probably be shocked to learn that their deleted data is still very visible on the disk.

Even after data has been overwritten on a disk it may still be accessible. Magnetic force microscopy and (MFM) magnetic force scanning tunneling microscopy (STM) are two techniques that enables the imaging of magnetization patterns with remarkably high resolutions. [4, 9] With MFM it is claimed that users can start getting images in 5 minutes, and with the knowledge of well documented disk formats and the identification of the correct location on a disk that an image could be developed in 2-10 minutes.[5]

Further it is claimed that a "reasonably capable" scanning probe microscope can be built that uses a PC as a controller and costs only about $US1400. The design plans for building the tools even are available on a web page.[9] The process of determining overwritten data could be automated thus further speeding the recovery of data of large disk drives.

These tools function by "peeling back" the layers of information that have been written to disk. Data is stored on a disk as patterns of varying magnetic strength and each write of the disk head changes the magnetic field strength at a position on the disk surface in a predictable manner.

The value of the magnetic field strength depends on the history of data stored in a given position. For instance writing a 1 over a 1 leaves a stronger magnetic signal then if a 1 had been written over a 0. By subtracting the existing data from the field strength signal one can determine the next underlying layer of data on the disk. Information can also be gleamed from the fact that the read write head of the disk experiences slight positional variation each time that data is written to the disk. A common analogy is made between writing a disk drive track and repainting the lines on the road. The new line that is laid down does not exactly travel the same path as the original. This leaves magnetic information at the sides of the track that can be used to read data that has been overwritten. A much more complete technical description of the process can be found in the references [4, 5].

Ones and zeros are actually stored on disk using various encoding patterns. These encoding patterns enable the recovery of a clock signal to identify the correct position of the magnetic signals on a disk surface. The encodings patterns are referred to as *run-length-limited encodings* (RLL) since they limit the number of consequetive ones and zeros appearing in the encoding patterns. Recent drives use a slightly different encoding scheme called *Partial-Response Maximum-Likelihood* (PRML) encoding. The difference is in the constraints placed on the encoding data patterns. Another older encoding scheme which is very similar to the RLL schemes is *modified RM* (MFM). Again a far more complete description of the encoding schemes can be found in the references [5].

To most effectively overwrite a portion of a disk we want to flip each magnetic domain on the disk as many times as possible. This requires the knowledge of the encoding pattern used by the disk drive. Ideally if users know the encoding schemes used by the drives they could simply select the overwrite patterns most effective for their drives. However since users probably will not have such knowledge we use overwrite patterns that should effectively flip all magnetic domains on current disk encoding schemes. These patterns are shown in Table 1. [5]

The ability to recover overwritten data from disks is somewhat questionable according to data recovery companies that were contacted by us for the purpose of this project. They indicated that data recovery after it has been overwritten even once was very difficult. Perhaps unsurprisingly they were also willing to try but suggested that it would cost "hundreds of thousands of dollars" and many months of work.

But the US Department of Defensive obviously believes that such data is recoverable easily enough since it specifies a standard, *DOD 5220.22-M* and referred to as "DOD standard 7-pass extended character rotation wiping", which mandates that 7 random read/write passes be made over data before it is considered securely destroyed. [8] It is generally believed that they have capabilities to recover data written over even more times. Various anecdotal stories can be easily found on the web. One author of this paper has heard such stories from interns at the FBI. It is difficult to ascertain what truly is possible though.

We provide our system with the caveat that our secure deletion tool uses the best know deletion techniques that still leaves the disk drive in a useable workable fashion. True peace of mind might come from only using a degaussing tool or following other suggested techniques such as burning, or pulverizing the disk.[5, 11]. Our secure deletion technique though is aimed at leaving the disk in a workable state. Anyone requiring complete guaranteed destruction of data and device obviously should pursue other options.

4 Implementation

The implementation of our ideas hopefully appear simple. We worked hard to come up with the most efficient implementation possible. As others have observed, modifying a file system is not an easy task. For this project we learned about the kernel implementations of signals, memory management,loadable kernel modules, the buffer cache, block device drivers, kernel locking primitives, the virtual file system layer, and obviously extensively about the ext2 filesystem. Overall our system currently entails roughly 900 lines or new or modified kernel code. This can be further reduced by carefully cleaning up the code particularly of debugging statements.

4.1 Secure deletion in Ext2 filesystem

The ext2 filesystem already contains an inode flags field for which one flag is supposed to indicate secure deletion. However secure deletion itself was not implemented probably because of the performance penalty. Various file flags can be listed using the `lsattr` command on Linux. Similarly flags can be set using `chattr`. The secure deletion flag is set by using the `chattr +s <filename>` command. Directories as well as files can be marked for secure deletion. Any files created in a directory marked for secure deletion will also have secure deletion flag set. This inheritance of attribute flags is beneficial since a user can mark entire trees in the filesystem name space for secure deletion where all the files created on the branches will have the secure deletion flag set. Flags are also preserved for all typical file operations. Users though should be aware that the copy command does not copy the file flags. (The new file would have the secure deletion attribute though if it was created in a directory marked for secure deletion.)

4.2 Overall System Design

The goal of our modifications are to completely remove the remnants of any deleted file or directory that had the secure deletion bit set. When a file is deleted or truncated we pass the released disk blocks to an asynchronous daemon process which overwrites

Data Written	Encoding Scheme Targeted
01010101 01010101 01010101 0x55	(1,7) RLL MFM
10101010 10101010 10101010 0xAA	(1,7) RLL MFM
10010010 01001001 00100100 0x92 0x49 0x24	(2,7) RLL MFM
01001001 00100100 10010010 0x49 0x24 0x92	(2,7) RLL MFM
00100100 10010010 01001001 0x24 0x92 0x49	(2,7) RLL MFM
00000000 00000000 00000000 0x00	(1,7) RLL (2,7) RLL
00010001 00010001 00010001 0x11	(1,7) RLL
00100010 00100010 00100010 0x22	(1,7) RLL
00110011 00110011 00110011 0x33	(1,7) RLL (2,7) RLL
01000100 01000100 01000100 0x44	(1,7) RLL
01010101 01010101 01010101 0x55	(1,7) RLL MFM
01100110 01100110 01100110 0x66	(1,7) RLL (2,7) RLL
01110111 01110111 01110111 0x77	(1,7) RLL
10001000 10001000 10001000 0x88	(1,7) RLL
10011001 10011001 10011001 0x99	(1,7) RLL (2,7) RLL
10101010 10101010 10101010 0xAA	(1,7) RLL MFM
10111011 10111011 10111011 0xBB	(1,7) RLL
11001100 11001100 11001100 0xCC	(1,7) RLL (2,7) RLL
11011101 11011101 11011101 0xDD	(1,7) RLL
11101110 11101110 11101110 0xEE	(1,7) RLL
11111111 11111111 11111111 0xFF	(1,7) RLL (2,7) RLL
10010010 01001001 00100100 0x92 0x49 0x24	(2,7) RLL MFM
01001001 00100100 10010010 0x49 0x24 0x92	(2,7) RLL MFM
00100100 10010010 01001001 0x24 0x92 0x49	(2,7) RLL MFM
01101101 10110110 11011011 0x6D 0xB6 0xDB	(2,7) RLL
10110110 11011011 01101101 0xB6 0xDB 0x6D	(2,7) RLL
11011011 01101101 10110110 0xDB 0x6D 0xB6	(2,7) RLL
Random	PMRL

Table 1: Optimum overwrite patterns for secure deletion of data under different disk encoding schemes

the data blocks a configurable number of times. Only after blocks have been overwritten do we return the blocks to the filesystem as free blocks to be reallocated. An important benefit of this approach is that the asynchronous daemon can perform the overwrites while the disk would otherwise be idle. Our approach sacrifices immediate security by allowing sensitive data to remain on the disk past the point where the user has asked for it to be deleted. However it ensures that regular disk operations can proceed without encountering too much increased disk access latency due to secure deletion of files. In the following sections we explain the modifications to the Ext2 filesystem code and the implementation of the asynchronous deletion daemon.

4.3 Ext2 Modifications

We had to make surprisingly few changes to the ext2 code itself to implement secure deletion. We made six modifications to the existing code base. These can be seen in Table 4.3. In each function we tested whether the secure deletion flag attribute was set. If it was not set then function behavior proceeded normally. However if the secure deletion flag was set then we modified the code to implement the overwriting process. When data blocks that need to be overwritten are released from a file, we add an element containing

ext2_free_blocks	modified to place blocks on daemon secure deletion list, call disk quota system freeing these blocks from the current inode, call disk quota system adding blocks back to the unique user and group inode
ext2_free_clean_blocks	renamed regular ext2_free_blocks
ext2_discard_prealloc	modified to call ext2_free_clean_blocks
ext2_delete_inode	modified to reset all inode values to zero
ext2_unlink	modified to ovewrite the name of a file from the directory
ext2_rmdir	modified to overwrite the name of the directory in the parent directory

Table 2: Modifications done to ext2 file system for implementing the secure deletion

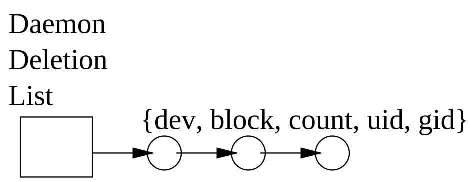

Figure 1: The deletion list data structure holding the data blocks removed from files

a tuple of device identifier, beginning block number, number of blocks released including this block number, the user identifier and the group identifier to a list called the daemon deletion list (Fig 1).

We pass the user and group identifiers to the deletion daemon so that the proper disk quota prospective can be maintained at all times. By proceeding in this fashion we only need to maintain one inode per user and gid pair. Otherwise we would need to keep all inodes alive for the entire time that blocks belonging to it were overwritten. We use this *uid, gid* identifier to maintain the number of blocks belonging to the pair currently being overwritten. Once the blocks have been overwritten we then call the disk quota system properly with a VFS inode dynamically created with the proper user, gid, device, and block count information. This does not interfere with the disk quota inode count since we are not freeing inodes but only disk blocks.

Currently the file metadata for securely deleted files and directories is overwritten once synchronously with all zeros within the Ext2 code itself. We made this decision since it was simpler to implement at the time. Later we intend to pass this information to the deletion daemon as well, so that it can be overwritten properly. For now though this is better then leaving file names, creation, access, modification, and deletion times accessible on the disk.

4.4 Ksdeletion Daemon

The Ksdeletion daemon starts at system initialization time or when the ext2 filesystem module is loaded. It is a kernel daemon that runs at periodic intervals, currently the default setting is every 10 seconds. Every time the daemon wakes up it performs three tasks: (1) retrieves and stores the blocks that need to be overwritten (2) issues new overwrite requests and (3) returns blocks that have been fully overwritten to the filesystem for reallocation.

Each device contains a generations data structure. Each generation contains two lists, one list for disk block information in the form of *block, count* groups and the second list containing user information in the form of *uid, gid* pairs. Different generations represent lists of blocks that have proceeded through the overwriting process a different number of times. The number of generations used depends upon the overwrite strategy used. If the policy is to completely overwrite a set of blocks multiple times before proceeding to the next set, then only two generations are required. One generation will represent the blocks currently being overwritten while the other generation will represent the blocks that have never been overwritten. Another policy though might mandate that blocks should be overwritten as soon as possible. To accommodate this type of policy we allow for a configurable number of overwrite generations. As blocks are progressively wiped they can be moved further in the generation data structure. New blocks are always added in the first generation slot. Figure 2 presents a diagram of this structure (our current implementation uses the second strategy where disk blocks are being progressively owner written over several generations). The daemon retrieves newly deleted blocks that the filesystem has added to the daemon deletion list and places them in the above data structure. The storage of the blocks on the daemon deletion list is inefficient since each *block, count* group requires its own list element. No aggregation of blocks across the entire device is possible in this format. But when the daemon adds the elements from the deletion list to the generation data structure, disk block information of adjoining disk blocks (belonging to the same generation) are merged to enable an efficient representation. If these cannot be merged a new element is created at the proper place in the list to ensure proper ordering of disk block numbers in the list. Since the list is ordered by the actual block number, it is easy to achieve very efficient overwriting of data especially when a large number of disk blocks are to be overwritten. We also add a *uid, gid, count* element to linked list of user information if the *uid, gid* pair is new, or increment the count of an existing *uid, gid* pair.

The second task of the daemon is to issue new write requests. Each time the daemon is activated it runs through all the devices writing out a configurable number of disk blocks using the overwrite patterns presented earlier. Before issuing write requests the daemon checks the number of outstanding device requests. If the number of outstanding requests (read or write) exceed a configurable level then no new requests are issued for that device. One side note is that the kernel interface to the block devices extensively make use of *buffer_head* data structures, however operations on the buffer head memory cache are not exposed. It would be beneficial if they were since then we could reuse the free buffer head pool already available. However since this is not exposed, we maintain our own cache which employ the underlying slab allocator memory routines. This is done to improve the efficiency of memory allocation.

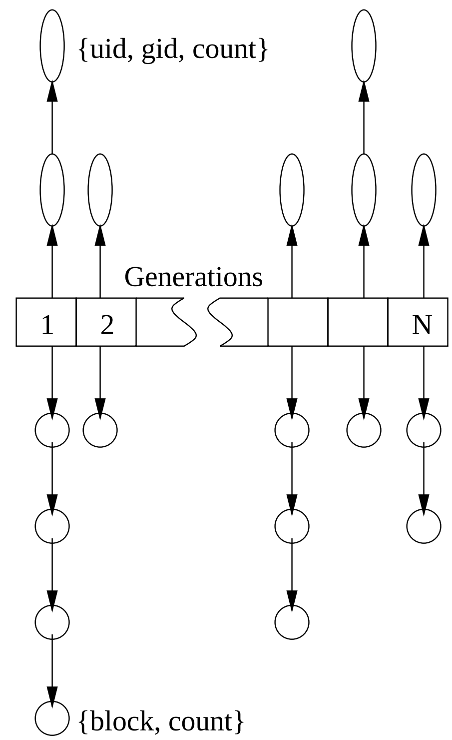

Figure 2: The generation data structure kept by secure deletion daemon

Finally the third task of the daemon is to return blocks which have been securely overwritten to the filesystem and update the disk quota system. All the blocks in one generation are returned to the free list at the same time. This enables us to use the list which maintained the *uid, gid, count* information to update the disk quota system properly.

4.5 Code details

We implemented our system using the latest Linux kernel sources which at the time was *linux-2.4test10*. The compiled daemon is $13KB$ in size however it con-

	INS	RES	WEB	NT	Sprite
hosts	19	13	1	8	55
users	326	50	7	8	76
days	31	31	24	31	8
data read (MB)	94619	52743	327838	125323	42929
data written (MB)	16804	14105	960	19802	9295
all events (thousands)	317859	112260	112260	145043	4602
unlink (thousands)	490	182	2	285	106
truncate (thousands)	37	4	0	1981	42

Table 3: Break down of file operations under typical working conditions for different file system implementations

tains many debugging statements. Also as suggested if kernel source maintainers were to place certain inlined functions into header files or if the buffer head cache was exposed our daemon would shrink in size even further.

5 Evaluation

In evaluating our system we were first curious to learn how often file deletion and truncation actually occurred in practice. We did not have time to collect actual data ourselves, however the study presented at USENIX 2000 (whose results table we partially reproduce in Table 3) suggests that file deletion and truncation are less then three percent of total file system activities. In most cases these activities constitute far less then one percent of disk operations. They do not characterize the amount of data deleted with each operation.

We did study how long user space programs took to overwrite a securely deleted file (Table 4). For this test we selected overwrite [14] which appears to be the best implementation of a user level secure deletion tool available. It uses Peter Gutmanns overwrite patterns to overwrite data thirty two times. The bandwidth of our disk drive as determined by the hdparm benchmark was $2.52 MB/sec$. (It is a relatively old IDE drive.). We content that even a relatively small file of $256KB$ which takes roughly 8 seconds to overwrite would be too much of a performance penalty to be acceptable to a user if used for frequently. Users would almost certainly not tolerate waiting 13 minutes for largest file.

We can determine the time taken to completely overwrite a file in our implementaion by using the following formula. Let:

T - time taken to securely overwrite the file
P - probability of being too busy when the daemon wakes up
F - file size in bytes
B - device block size in bytes
N - maximum number of blocks written for iteration
G - number different overwriting generations
I - time between iterations
Then;

$$T = G \times I \times \frac{1}{P} \left\lceil \frac{\lceil \frac{F}{B} \rceil}{N} \right\rceil$$

Had we had more time we would have liked to explore whether disk fragmentation is increased as a result of our holding blocks for the overwriting process or whether background disk writes have any effect on other user disk activity times. Conceivably seek times could be increased since the disk read write head will be moved more frequently to other sections of the disk. Other pertinent questions include the effect of our overwriting process on disk lifetimes since we are increasing the amount of data written to disk. (We suspect though that this would not be much of a problem given the infrequency of file deletion and truncation suggested earlier.) We also would have liked to have more completely explored different overwrite scheduling policies.

File Size	Elapsed Time
1KB	1.20 secs
2KB	1.24 secs
4KB	1.87 secs
8KB	1.88 secs
16KB	1.96 secs
32KB	2.81 secs
64KB	3.71 secs
128KB	5.14 secs
256KB	8.74 secs
512KB	15.18 secs
1024KB	28.51 secs
2MB	54.48 secs
4MB	1:43 mins
8MB	3:25 mins
16MB	6:49 mins
32MB	13:27 mins

Table 4: Secure file deletion times for a user level deletion mechanism

6 Conclusion

This work presents a secure data deletion mechanism for the Linux ext2 filesystem. We asynchronously overwrite data according to best known practices. We securely overwrite both the file data and meta data. Our system is configurable to user needs and the overwriting takes place during periods when disk would otherwise be inactive. We have argued why existing approaches such as user level deletion programs and encryption based approaches are inadequate. Perhaps the most compelling argument for our approach though is that it is very simple from a users standpoint. All a user has to do to securely delete their data is set a flag bit on the file or the directory containing that file.

Some work remains to be done to ensure that sensitive data never remains behind on the disk. The main task is to manage overwriting of sensitive data in the swap disk. The easiest solution would be to ensure that it never gets written to the swap disk in the first place. All memory that contains sensitive data could be *mlocked*. However this is probably not an adequate solution. Therefore current plans are to let the data be written to the swap disk and periodically overwrite the swap disk while obviously moving swap data around appropriately. We still have to think about this though and indeed learn about how the swap disk space is actually managed. We also have to modify the `fsck` facility to clean the disk blocks that are found by the system. We are currently in the process of beginning these modifications.

Other things to consider as we develop our system more fully are the primitives that the disks actually offer that we can employ to our advantage. For instance being able to control the frequency and intensity of disk head signal as the overwriting overwrite patterns are written may improve the overwrite effectiveness. [6] We also might considerably improve overwrite efficiency by using rotational latency periods for our overwrite passes. [7]

We are aware that this tool potentially benefits people that we might not really want to help. In discussing this project with colleagues, concerns were raised that it would help hackers hide their activities and allow criminals to evade the law more easily by removing potential evidence. In many ways these concerns are similar to objections about the wide spread use of encryption. The technology can be used for both good and bad purposes. We strongly suspect though that it will give piece of mind to far more people then the number who will use it to evade the law. Similarly we recommend to those concerned with the human impact of such a tool the article *"In Defense of the DELETE Key"*. [12] It convincingly argues that the disk drive is an electronic recording device present in every office and home that records our very written thoughts. If a user changes their mind later and deletes what they wrote, the file actually should be deleted. The author provides convincing examples of why everyday users would actually want DELETE to really mean delete.

One final caveat to the user would be that we really do not know how securely we have actually deleted data from a file system. We use the best known secure practices of today. However years in the future this information could be recovered using more advanced techniques. Perhaps just as required key length has increased for encryption the number of overwrite passes for securely deleting data will also need to be increased. We are reminded of this fact recently upon learning that the deleted sections of the famous Watergate tapes could now be recovered.

Finally we intend to clean up our code, add many required error checking elements, and then release the code to the Linux community. We hope to be able to convince the maintainers of the ext2 filesystem that they should integrate our deletion daemon into the main code body. Since ext2 is included with the Linux kernel sources we hopefully will have contributed something useful to the larger free Unix community.

References

[1] "Andrew File System (AFS)," http://www.cs.cmu.edu/afs.html

[2] "CyberScrub," http://www.cyberscrub.com/

[3] "Ext2," http://e2fsprogs.sourceforge.net

[4] R. Gomez, A. Adly, I. Mayergoyz, E. Burke, "Magnetic Force Scanning Tunnelling Microscope Imaging of Overwritten Data", IEEE Trans.on Magnetics, Vol.28, No.5 (September 1992), p.3141.

[5] P. Gutmann, "Secure Deletion of Data from Magnetic and Solid-State Memory," Proceedings of Sixth USENIX Security Symposium, San Jose, CA, 1996.

[6] T. Lin, J. Christner, T. Mitchell, J-S Gau, and P. George, "Effects of Current and Frequency on Write, Read, and Erase Widths for Thin-Film Inductive and Magnetoresistive Heads",IEEE Trans.on Magnetics, Vol.25, No.1 (January 1989), p.710.

[7] C. Lumb, J. Schindler, G. Ganger, D. Nagle, E. Riedel, "Towards Higher Disk Head Utilization: Extracting Free Bandwidth From Busy Disk Drives," Proceedings of OSDI, 2000.

[8] National Industrial Security Program Operating Manual (NISPOM), DoD 5220.22-M, January 1995.

[9] "NanoTools: The Homebrew STM Page", http://atom.snu.ac.kr/stmwebpage.html

[10] D.Rugar, H.Mamin, P.Guenther, S.Lambert, J.Stern, I.McFadyen, and T.Yogi, "Magnetic force microscopy: General principles and application to longitudinal recording media", Journal of Applied Physics, Vol.68, No.3 (August 1990), p.1169.

[11] "Magnetic Tape Degausser", NSA/CSS Specification L14-4-A, 31 October 1985.

[12] J. Rosenbaum, "In Defense of the DELETE Key," The Green Bag, 2D Series Vol 3, No. 4, Summer 2000.

[13] D. Roselli, J. Lorch, T. Anderson, "A Comparison of File System Workloads," Proceedings of USENIX Annual Technical Conference, 2000.

[14] "Overwrite, Secure Deletion Software," http://www.kyuzz.org/antirez/overwrite/

[15] "Shred-X", http://www.bsoft.ic24.net/shredx.htm

[16] "Wipe homepage," http://wipe.sourceforge.net/

[17] J-G Zhu, Y. Luo, J. Ding, "Magnetic Force Microscopy Study of Edge Overwrite Characteristics in Thin Film Media", IEEE Trans.on Magnetics, Vol.30, No.6 (November 1994), p.4242.