USENIX Association

Proceedings of the 10th USENIX Security Symposium

Washington, D.C., USA
August 13–17, 2001

© 2001 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Secure Data Deletion for Linux File Systems

Steven Bauer, Nissanka B. Priyantha
{bauer,bodhi}@lcs.mit.edu
MIT Laboratory for Computer Science

Abstract

Security conscious users of file systems require that deleted information and its associated meta-data are 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 meta-data 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 kernel for asynchronous secure deletion of file data and meta-data. We provide an implementation for the *Ext2* file system; other file systems can be accommodated easily. 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 in many respects and that synchronous deletion facilities are too time consuming to be acceptable to users. Further, we contend that encrypting file information, either using manual tools or a encrypted file system, is not a sufficient solution to alleviate the need for secure data deletion.

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 completely and in a truly usable fashion. After we explain our original motivation and background material, 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 remote locations including home and office machines, terminals in airports and Internet cafes, and multiple workstations in universities' computer clusters. Many applications and distributed file systems cache sensitive user data on the local disks to improve performance. Distributed file systems such as AFS [11] may cache user data on the client machine or users themselves may copy sensitive files to the local directories. Many web browsers cache accessed information on the local drive. Even when the web cache is cleared the data remains on the underlying physical disk.

Users need assurances that their sensitive data does not remain visible on every machine they use. Sensitive data must be overwritten, sometimes with multiple overwrite passes, making the original data inaccessible even to advanced probing techniques. We have implemented such a secure deletion mechanism in the form of a configurable kernel daemon that asynchronously overwrites disk blocks. The interface to the daemon is general; any block-oriented file system can use the daemon to overwrite blocks on a particular device. Once the overwrite process is complete, the daemon invokes a registered callback that updates the file system state.

Section 2 briefly covers 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 requirements for ensuring that deleted data cannot be recovered. Section 3 discusses the shortcomings of user-level secure deletion tools and problems with relying solely upon cryptographic techniques to prevent deleted data from being accessible. Section 4 details the design goals of our system. Section 5 details our implementation and usage suggestions. Section 6 considers

the performance, ease of use, applicability, and security of our system. Finally, Section 7 concludes the paper.

2 Background

Recovering data deleted normally from a disk drive is remarkably easy. Most users are not aware that after they delete a file it still remains visible on the disk until overwritten by new data. This may mean deleted data remains on the disk for considerable lengths of time. Many user space tools that recover deleted files exist both for Unix, Windows and Macintosh machines[8]. The ability to recover supposedly deleted files is beneficial to users who inadvertently remove important files, but most people would be shocked to learn that their deleted data is still very accessible.

Even after data has been overwritten it may still be accessible. Magnetic force microscopy and (MFM) magnetic force scanning tunneling microscopy (STM) are two techniques that enable the imaging of magnetization patterns with remarkably high resolutions. Using MFM and a knowledge of well documented disk formats an image can be developed in under ten minutes for older drives[10]. 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 in a predictable manner. Scanning tools can "peel back" layers of this magnetic information recovering older data. (A much more complete technical description of the process can be found in the references[10, 12].)

On older disks the encoding patterns are referred to as *run-length-limited encodings* (RLL) since they limit the number of consecutive ones and zeros appearing in the encoding patterns. Modern drives use a different encoding scheme called *Partial-Response Maximum-Likelihood* (PRML) encoding. The difference is in the constraints placed on the encoding data patterns. To most effectively overwrite a portion of a disk each magnetic domain on the disk should be flipped a number of times. While older drives could be overwritten more effectively by employing particular overwrite patterns, specific patterns have not been designed for the existing PRML encoding techniques.

There is considerable controversy regarding the capability to recover data that has been overwritten. Prevailing attitudes among some Internet communities seems to be that various government agencies have the ability to recover data from drives even if the data has been overwritten *any* number of times. Numerous anecdotal stories regarding these supposed capabilities can be found on the web. Though still heavily referenced, the Department of Defense standard *DOD 5220.22-M* [16] is outdated and does not reflect current drive technology. It mandates that seven random read/write passes be made over data before it is considered securely destroyed. This compels many users to believe that large numbers of overwrite passes are required.

Twenty commercial data recovery companies were contacted during this project. Each was asked if they could recover a 100KB data file that had been accidentally completely overwritten once with random data. All but one indicated that they could not recover the data if it had really been overwritten. One company [2] that did possess appropriate tools and was willing to try casually estimated the chance of success at "less than one percent."

It is difficult to ascertain what truly is possible. Given the wide variety of opinions and desires, our approach is to let users select the deletion procedures they feel most comfortable using. We strongly suspect that users will have made the recovery task impossible with a small number of overwriting passes at least for modern disk drives. For older drives, additional overwrite passes should be employed[10].

3 Limitations of Alternative Approaches

User-level secure deletion tools and cryptographic techniques attempt to provide some guarantees that sensitive data will not be recoverable once it is deleted. These approaches are useful in many respects, but are not always an appropriate solution. Secure data deletion implemented at the file system level is still required. This section details the limitations of user space tools and cryptographic techniques for secure data deletion.

3.1 User Space Deletion Tools

Any implementation of secure deletion at the user-level will be inadequate. Numerous user-level secure deletion tools already exist (for example [5, 17, 19, 21].) While these tools can be used productively for immediate synchronous overwriting of individual files, they do not provide a complete usable solution. All user space programs face the following problems that cannot be addressed effectively.

1. **File meta-data cannot be overwritten completely at the user level:** The sensitive information in a file system includes the contents of a file and the file meta-data including the name, size, owner information, and creation, modification, access, and deletion times. User-level programs overwrite only the file data itself. Although parts of the meta-data could be overwritten using `touch` to set the file access, creation, and modification times and renaming the file to obfuscate the file name, such techniques are cumbersome and inefficient. Even with such a workaround, important file meta-data information such as what blocks the file contained, user and group ownerships, and deletion time could not be removed.

2. **Secure deletion tools cannot be interposed between all 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 means 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, and particularly problematic, a user-level deletion tool would have to be integrated with every application that handles its own file management. While dynamically linked libraries could be changed appropriately, statically linked binaries would remain a problem.

3. **File truncation cannot be handled effectively:** User-level deletion programs only overwrite an entire file and thus do not handle file truncation. If someone uses an editor to delete half a file, causing some blocks to be returned to the list of free blocks, 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 and is inefficient if the truncation would not actually return a block to the free list. Fundamentally applications calling `truncate` do not have enough information to know if blocks have to be overwritten. Again, all applications invoking file truncate would have to be modified.

4. **Overwriting data synchronously is inconvenient and often unusable 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 after the deletion tool is called. The deletion tool could rename a file before the overwrite process begins; however, renaming is not possible for a truncation operation since the file is not removed from the name space.

3.2 Cryptographic Techniques

Another approach to prevent user data from remaining visible on the disk is to use encryption. The possible approaches are to use either an encrypted file system or encryption tools to selectively encrypt files on a disk. The idea is that encoded data will not be accessible without the encryption key. If the encryption key can be intentionally lost, the data is destroyed without overwriting it.

This approach has been proposed [4] for revoking data from both the file system and all backup tapes on which the data is stored. The security of this particular system depends upon either the proper management and destruction

of personal copies of a master key or upon a third party trusted to properly destroy their public private key pairs periodically. While being a compelling solution from the standpoint of handling all backup data, the solution is not as satisfying from a practical standpoint. Users who want data to be securely deleted may not want to trust *any* third parties and the alternative of storing and destroying personal copies of encryption keys will seem laborious to many users. We address how backups should be handled in our system in Section 7.

Another cryptographic approach is to use a Steganographic File System[1]. While not attempting to achieve the same goal, deleted data in such a system could not be proved to exist, therefore a user could not be compelled to turn over an encryption key. Secure deletion must mean that even the owner of the data cannot recover it later by any means. There is a psychological need satisfied by knowing that even oneself cannot recover securely deleted data.

In a more general sense, all cryptographic approaches suffer from the following common problems.

1. **Encryption keys can be revoked or compromised:** Anytime that a key is revoked the data associated with it must be considered accessible. Users can be compelled by law to turn over their encryption keys or their keys can be otherwise compromised. In any case, deleted encrypted-files that remain on a disk are as accessible as plain-text if the key is available.

2. **Encryption may not be a viable option for performance or legal reasons:** The performance of an encrypted file system compared to a regular file system may be unacceptable. Encryption tools may be too much of a performance penalty since the tool must be used for every file operation, not just at data deletion time. Users may simply want assurances that their deleted data is not accessible but not want all file data to be encrypted. In some countries, encrypting file data may not be a legal option or available encryption may not be acceptably strong.

3. **Plaintext files may remain visible:** Before files are encrypted, they may have existed as plaintext stored either in temporary or regular files. See the BugTraq archives [3] for an interesting discussion of plaintext temporary files created by the Windows 2000 EFS [14] that remain visible on disk in some common cases. These plain text files must be deleted securely.

4 Design Goals

Our design focused on addressing two main goals: *completeness* and *transparency*. Completeness entails that all the data over the entire course of a file's existence must have been securely overwritten. File data read into application buffers and swapped out to a swap disk must not be accessible either. Security guarantees must be maintained if a system crash happens before the disk blocks have been fully overwritten. 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 the procedures followed after crash need to be modified to handle this problem. The goal is for no remnants of a deleted file to remain anywhere on the physical disk.

Our completeness requirement 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 large in-memory 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 remap blocks to other sectors upon sector failure must be addressed. Another issue that must be considered is the possibility of having multiple copies of a file, either because of a users' actions or the backup policies of a system administrator. While our system does not address this facet of completeness, we do our best in our system documentation to ensure that users are 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, they do not consider the effects of the underlying disk blocks being returned to a free list. 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 is inherently a slow operation that involves ensuring the underlying disk blocks have been overwritten with data multiple times. Overwriting a file with even two passes of data involves writing the data, flushing the data to disk, waiting for the head seek and writing to complete, and then repeating the process. This will be unacceptably long for even a small file. Were a user to delete a file with megabytes of data, the removal process would be intolerably 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 that 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 be maintained properly during the period of time while the blocks are being overwritten. We require that a user's disk quota reflect the fact that the disk blocks being overwritten are not available for reassignment. Blocks remain a part of a user's 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.

We provide our system with the caveat that our secure deletion tool intentionally uses deletion techniques that preserves the integrity of the disk drive. For some users, true peace of mind may come only from using a degaussing tool or following other suggested techniques such as burning, or pulverizing the disk[7]. Our secure deletion technique is designed for those interested in leaving their disk in a working state. Anyone requiring more extensive destruction of data and device obviously should pursue other options.

5 Implementation

Our system is split into two parts, a kernel daemon that overwrites blocks on a device, and modifications to a file system that hands off disk blocks to the daemon and appropriately overwrites file meta-data. This is a general design that can support any block-oriented file system. We have implemented the necessary modifications for the Linux Ext2 file system. Other file systems may be added as time permits.

The goal of our modifications is to completely remove the remnants of any file or directory that needs to be deleted securely. When a file is deleted or truncated, we pass the released disk blocks to an asynchronous daemon process that overwrites the data blocks a configurable number of times. Only after the blocks have been overwritten do we return them to the file system 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 deleted it. However it ensures that regular disk operations can proceed without being delayed by the secure deletion of files. In the following sections we explain the modifications to the Ext2 file system code and the implementation of the asynchronous deletion daemon.

We implemented our system using the latest Linux kernel version, which at the time was *linux-2.4.2*. The compiled daemon is $12KB$ in size. The modifications to the file system adds an additional 3900B to Ext2 file system. Overall the code for the overwriting daemon entails roughly 800 lines of kernel code. We have been successfully running this system for the past month on one of the author's machines.

5.1 Secure deletion in Ext2 file system

The Ext2 file system already contains an inode flags field for which one flag is supposed to indicate secure deletion. Secure deletion itself was not previously implemented probably because of the performance penalty of a naive im-

plementation. On Linux, various file flags can be listed using the `lsattr` command and set using `chattr`. The secure deletion flag is set by issuing the

```
chattr +s <filename>
```

command. Directories as well as files can be marked for secure deletion. Any file created in a directory marked for secure deletion will have the secure deletion flag set. This inheritance of flags is beneficial since a user can mark entire trees in the file system name space for secure deletion where all files created on the branches will have the secure deletion flag set. Flags are preserved for most typical file operations. Users should be aware that the copy command does not copy the file flags. (The new file would have the secure deletion flag set if it was created in a directory marked for secure deletion.)

5.2 Ext2 Modifications

We had to make few changes to the Ext2 code to implement secure deletion. We made seven modifications to the existing code base. These can be seen in Table 1. In each function we tested whether the secure deletion flag was set. If it was not set then function behavior proceeded normally. 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 a tuple of device identifier, beginning block number, number of blocks released including this block number, the user and group identifiers, and the function to be called once the blocks have been overwritten to the daemon deletion list (Figure 1).

We pass the user and group identifiers to the deletion daemon so that the correct disk quota can be maintained at all times. After the disk block has been returned to the free list, the disk quota system is updated using these identifiers. We use the *(uid, gid)* pair 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 with a VFS inode dynamically created with the proper user, group, device, and block count information. This does not interfere with the disk

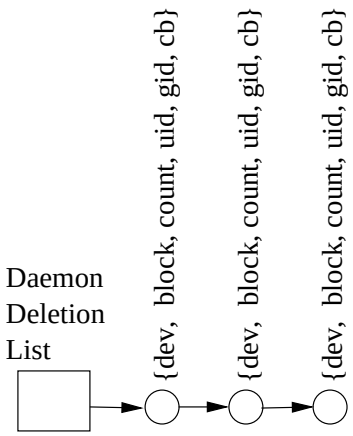

Figure 1: The deletion list data structure holding the data blocks removed from files

quota inode count since we are not freeing inodes but only disk blocks.

In our current implementation 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. Also, currently between the time when blocks are initially passed to the deletion daemon and the time they are returned to the free list, the blocks are not associated with any file in the file system. This complicates the recovery process if a crash occurs during this time. Future modifications will add the blocks to files in a directory storing all the blocks that must be overwritten.

5.3 Ksdeletion Daemon

The Ksdeletion daemon starts at the system initialization time or when the secure deletion module is loaded. It is a kernel daemon that runs at periodic intervals. Behavior of the daemon is dynamically configurable through a `/proc` file system interface. 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 corresponding file system for reallocation. A file system is prevented from being unmounted if its blocks are in the process of

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 overwrite the name of a file from the directory
ext2_rmdir	modified to overwrite the name of the directory in the parent directory
ext2_free_overwritten_blocks	new function, returns overwritten blocks to the free list and updates disk quota

Table 1: Modifications done to Ext2 file system for implementing the secure deletion

being overwritten.

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 employed. 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 might mandate that blocks 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 overwritten they are moved further in the generation data structure. New blocks are always added in the first generation slot. Figure 2 presents a diagram of this data structure.

The daemon retrieves newly deleted blocks that the file system 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 a new set of blocks cannot be merged, a new element is created at the proper place in the list to ensure the correct ordering of disk block numbers. 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 a 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 device write requests. Each time the daemon is activated it runs through all the devices writing out a configurable number of disk blocks. Before issuing write requests the daemon checks the number of outstanding device requests. If the number of outstanding requests (read or write) exceeds 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 makes use of *buffer_head* data structures, but 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. Since this is not exposed, we maintain our own buffer head cache that employs the underlying slab allocation memory routines. This is done to improve the efficiency of memory allocation.

Finally, the third task of the daemon is to return blocks that have been securely overwrit-

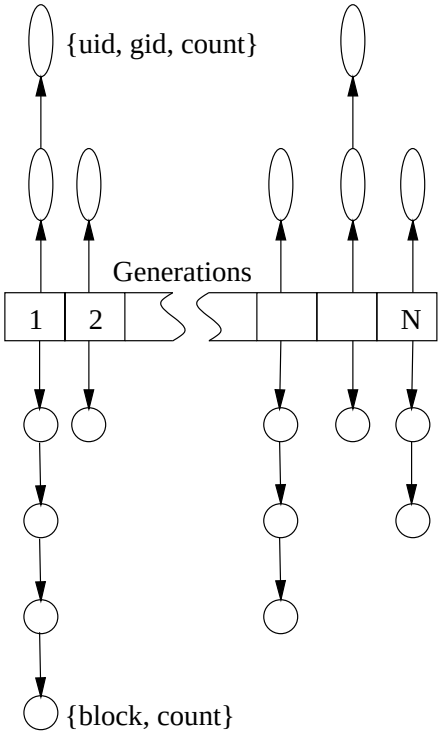

Figure 2: The generation data structure kept by secure deletion daemon

ten to the file system by invoking the registered callback of the file system. All the blocks in one generation are returned to the free list at the same time. The disk quota system then is properly updated using the *(uid, gid, count)* list.

5.4 Overwrite Policies

Currently the daemon overwrites a configurable number of blocks on each device for each iteration. Another policy being implemented is to overwrite either a minimum number or a percentage of outstanding blocks, whichever is larger. Other policies that could be implemented include overwriting blocks at a rate determined by the amount of free space remaining or policies that make more sophisticated attempts at avoiding regular disk activity.

Parameters of an overwrite policy are dynamically configurable through the `/proc` file system. For the default policy, parameters include maximum number of blocks written at a time, overwrite patterns employed, number of overwrite passes, and the interval between passes. These parameters are set by writing commands of the following form to `/proc/secdel/policy`:

```
secdel <device> <parameter> <value>
```

5.5 Failure Recovery

A file system may crash before data blocks are overwritten and returned to the free list or sensitive data may be written to a block but not yet be pointed to by a file system element. In either case the inconsistencies that appear in the file system are blocks and inodes that are marked as allocated but are not linked to the rest of the file system. After such a crash the system administrator determines if allocated blocks should be securely overwritten or if they belong to existing files. While still needing to handle blocks that are written but not yet pointed to by a file system, our planned modifications, adding blocks being securely overwritten to temporary files, will make the recovery task easier.

5.6 Backup Copies

We imagine that many users will not need to worry about additional copies of their sensitive data. In many cases backup mechanisms are not employed. In other instances, no backup procedure is necessary since user data is only temporarily accessed and stored on the device (for instance on terminals in Internet cafes or academic computing clusters.) If a user does make backup copies, we suggest that separate devices be used to store data which may need to be securely deleted. Other handling procedures should be instituted for these devices; for instance, backup tapes may be destroyed after a shorter time period.

5.7 Swap Space

A number of approaches exist so that sensitive data is not stored in the swap space on a disk. The easiest solution is to ensure that sensitive data never gets written to the swap disk in the first place. Given the availability of large amounts of memory, the swap-file could simply be disabled. Slightly more complex, an encrypted swap space using a random key could be used. If a longer window of vulnerability is

tolerated, the swap-file could be disabled, overwritten and then re-enabled periodically. devices

6 Evaluation

We evaluate our security mechanism using a number of criteria including performance, ease of use, applicability and security. Each of these criteria is important considering the target population for our tool.

6.1 Performance

The impact on user visible latency of our kernel daemon is negligible. Awakened at periodic intervals, the daemon returns immediately if no blocks need to be overwritten. Similarly, the performance impact of our modifications to the Ext2 file system is not detectable by our tests. Measurements of the time taken to return from a user space deletion and truncation were not affected by the addition of our code.

According to [6] file deletion and truncation are less than three percent of total file system operations on a variety of system platforms and workloads. In most cases, these activities constitute far less then one percent of disk operations. Their study does not characterize the amount of data deleted with each operation which is pertinent to our overwrite process. This still tends to indicate that our daemon will be inactive for a majority of the time.

We did not explore whether disk fragmentation is increased as a result of 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. Another pertinent performance question is the effect of our overwriting process on disk lifetimes since we are increasing the amount of data written to disk. We suspect that this would not be much of a problem given the infrequency of file deletion and truncation suggested by[6].

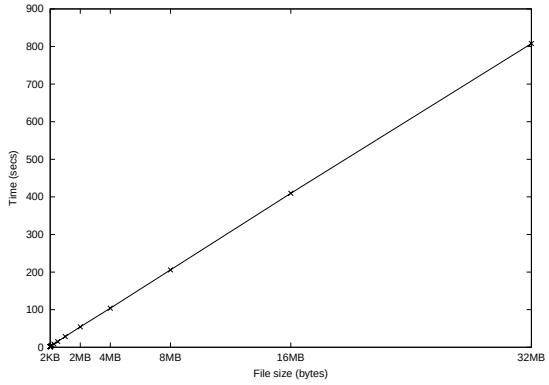

Figure 3: Secure file deletion times for a user-level deletion tool

6.2 Ease of Use

Our secure deletion tool is relatively easy to install since both the daemon and the modified file systems can be loaded as kernel modules. A new kernel does not need to be recompiled. Overwrite policies and their associated parameters can easily be configured through the `/proc` file system interface. Users then only need to make a one-time decision regarding which directories or files should have their blocks securely deleted. In Ext2, entire directory trees can be flagged for secure deletion by issuing one command. From a user perspective, all file operations and application behaviors proceed normally.

To compare the ease of use of our tool with user space tools, we examined how long a typical user space program took to overwrite files of various sizes (Figure 3). For this test, we selected `overwrite` [17], which appears to be a popular secure deletion tool. It uses Peter Gutmann's overwrite patterns [10] to overwrite files thirty-two times. The bandwidth of our disk drive as determined by the hdparm benchmark was $2.52MB/sec$. (It is a relatively old IDE drive.) We believe that even a small file of $256KB$, that takes roughly 8 seconds to overwrite, is too much of a performance penalty to be acceptable if employed frequently. Users definitely would not tolerate waiting 13 minutes for larger file sizes to be deleted.

6.3 Applicability

Any block-oriented file system can use the asynchronous overwrite functionality provided by our deletion daemon to securely delete blocks of data. Log structured file systems, file systems which write redundant data, such as RAID-based file systems, and drives which hide failures through dynamically remapping blocks are not supported. Systems employing backup strategies were addressed in a previous section. We provide warnings with our system documentation so that users are aware when this mechanism should not be utilized.

6.4 Security

The security of our approach depends on the effectiveness of the overwriting techniques employed and the window of opportunity between when the request arrived for the secure deletion and when the overwriting process actually completes. While we strongly suspect that even a small number of overwrite passes will make recovery of deleted data impossible for most modern drives, we cannot know for sure. Users who feel they need more security can configure their installation to employ more overwrite passes.

We can approximate the time taken to completely overwrite a file in our implementation using the following formula assuming that writes from one iteration are complete before the next iteration begins.

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 per iteration*
G - *number of different overwriting passes*
I - *time interval between issuing overwrites*
Then;

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

Fig 3 presents sample overwrite times assuming a device block size of 4096B, 1024 blocks written per iteration, a 10 second interval between overwrites, and a zero probability of the request queue being too busy when the daemon checks. Assuming a zero probability of being too busy makes the results presented a best-case scenario. The overwrite policy employed is to completely finish one overwrite pass before proceeding to the next. This results in the step behavior demonstrated in the graph.

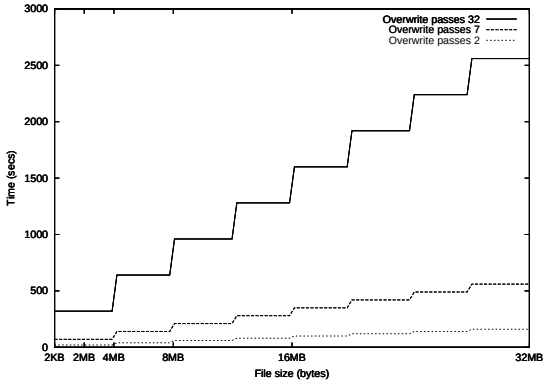

Figure 4: Time to securely delete files for number of overwrite passes

7 Conclusion

This work presents a secure data deletion mechanism for the Linux Ext2 file system. 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 can take place during periods when the 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 is that it is very simple from a user's standpoint. All users have to do to securely delete their data is to set a flag bit on the file or the directory containing that file.

Other things to consider as we develop our system more fully are the primitives that the device drivers offer that we can employ to our advantage. For instance, being able to control the frequency and intensity of disk head signal as the patterns are written may improve the overwrite effectiveness[20]. We also might considerably improve the overwrite efficiency by using rotational latency periods for our passes[13]. Another possible implementation revision is to

employ the techniques described in [9] for updating file system meta-data.

We are aware that this tool potentially benefits people who we might not be interested in helping. 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 providing readily available tools for removing evidence both on their own and compromised systems. 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 that it will give peace 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"*[18]. It argues that the disk drive is an electronic recording device present in every office and home that records all our written thoughts. If users change their mind later and delete what they wrote, the file actually should be deleted. The author provides convincing examples of why everyday users would want 'delete' to really mean delete.

One final caveat to the user would be that we do not know how securely we have removed data from the physical disk. We use the best known practices available today. Years into the future, this information potentially could be recovered using more advanced techniques. As recording mediums change, the methods for securely deleting data may need to be modified. We were reminded of this fact recently upon learning that the National Archives was investigating whether the deleted sections of the famous Watergate tapes could now be recovered[15].

8 Availability

Our code, installation and usage instructions is available under GPL at the following location:

`http//atlas.lcs.mit.edu/securedeletion`

References

[1] Anderson, Needham, and Shamir. The Steganographic File System. In *IWIH: International Workshop on Information Hiding* (1998).

[2] Authentec International. Private communications.

[3] Berglind, R. BugTraq: EFS Win 2000 flaw. Post to bugtraq mailing list. http://archives.linuxbe.org/arch051/0037.html, January 2001.

[4] Boneh, D., and Lipton, R. A Revocable Backup System. In *Proceedings of the Sixth USENIX Security Symposium* (1996).

[5] CyberScrub Secure File Deletion / Internet Privacy Utility. http://www.cyberscrub.com.

[6] D. Roselli, J. Lorch, T. A. A Comparison of File System Workloads. In *Proceedings of USENIX Annual Technical Conference* (2000).

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

[8] Ext2fs Home Page. http://e2fsprogs.sourceforge.net/ext2.html.

[9] Ganger, G. R., McKusick, M. K., Soules, C. A. N., and Patt, Y. N. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems 18*, 2 (2000), 127–153.

[10] Gutmann, P. Secure Deletion of Data from Magnetic and Solid-State Memory. In Proceedings of the Sixth USENIX Security Symposium, pages 77-89, 1996.

[11] Howard, J. H. An overview of the andrew file system. In *Proceedings of the USENIX Winter 1988 Technical Conference* (Berkeley, CA, 1988), USENIX Association, pp. 23–26.

[12] J-G Zhu, Y. Luo, J. D. Magnetic Force Microscopy Study of Edge Overwrite Characteristics in Thin Film Media. *IEEE Trans.on Magnetics 30*, 2 (1994), 4242.

[13] Lumb, C., Schindler, J., Ganger, G., Nagle, D., and Riedel, E. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Symposium on Operating Systems Design and Implementation* (October 2000).

[14] MICROSOFT. Encrypting File System for Windows 2000. http://www.microsoft.com/windows2000.

[15] NARA. NARA News Release 00-105. http://www.nara.gov/nara/pressrelease/nr00-105.html.

[16] National Industrial Security Program Operating Manual. DoD 5220.22-M, January 1995.

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

[18] In Defense of the DELETE Key. *The Green Bag 3*, 4 (2000).

[19] Shred Info Entry. Linux INFO tree, File: fileutils.info Node: shred invocation.

[20] T. LIN, J. CHRISTNER, T. M. Effects of Current and Frequency on Write, Read, and Erase Widths for Thin-Film Inductive and Magnetoresistive Heads. *IEEE Trans.on Magnetics 25*, 1 (1989), 710.

[21] Wipe: Secure File Deletion. http://wipe.sourceforge.net/.