Adding Secure Deletion to Your Favorite File System

Nikolai Joukov and Erez Zadok
Stony Brook University

Appears in the proceedings of the third international IEEE Security in Storage Workshop (SISW 2005)

Abstract

Files or even their names often contain confidential or secret information. Most users believe that such information is erased as soon as they delete a file. Even those who know that this is not true often ignore the issue. Nevertheless, recovering deleted files is trivial and can be performed even by novice hackers. The problem is exacerbated by the widespread of portable and mobile storage devices. This type of unwanted after-deletion data recovery is in part an education problem. Users believe that deleted files are erased, even though they are not. Retraining and educating users is difficult. Therefore, storage systems should behave appropriately—the data should be erased from the storage on a per-delete basis.

We found that existing solutions are either inconvenient, inefficient, or insecure. We have designed Purgefs: a file system extension that transparently overwrites files on the per-delete basis. Purgefs can be *automatically* added to a number of existing and *future* file systems, including networked and stackable file systems. Purgefs supports multiple policies to trade-off performance with the level of purging guarantees. We demonstrate that Purgefs does not add overheads or perturb users' activity under typical user workloads.

Keywords: Security, Unintended data recovery

1 Introduction

After a file is deleted, its data is still stored on the physical media until the actual data blocks are overwritten. Sometimes, this allows users to recover a file they mistakenly deleted. Unfortunately, a malicious user can also recover such a deleted file. A local privileged user can access a low-level device via the `/dev` interface and use one of several available software products to get the whole file or portions of it [6]. Even worse, floppy, flash, rewritable optical, and hard disks can be discarded, lost, or stolen. In this case a malicious person can recover the data using a local machine and a local privileged account. Overwriting the erased data or a whole disk prevents this simple method of data recovery, which requires no extra hardware.

More sophisticated tools can potentially recover the data even if the data is overwritten [14]. For example, disk head positioning over a track is not perfect. A single data overwrite operation is likely to overwrite a track that is slightly misaligned relative to the original data track. This means that even after a track is overwritten, there may exist a narrow portion of the track with the original information intact. Therefore, hardware tools can be used to read the narrow track of data that was not overwritten. Overwriting the data multiple times makes such a recovery more difficult.

This after-delete data recovery is counterintuitive to most users. A majority of users want deleted files to be permanently erased, and they believe that deleted files are physically erased [22]. This can lead to the situation that media with financial or otherwise sensitive information can be treated as cleared and "safe-to-lose" by the users. For example, every third hard drive resold on eBay contains confidential information such as credit card or medical records [7]. That is why government agencies and some businesses require proper sanitation of the physical media that was used to store sensitive information [9, 23]. However, as in the case of many security-related problems, security must be balanced with convenience and performance. Therefore, solutions to the problem vary for different situations. Generally, there are six solutions to the problem of recovery of deleted data:

1. Magnetic media can be degaussed; more generally the storage media can be destroyed. In terms of possible data recovery, this is the most reliable data-erasure method. Unfortunately, this solution is either expensive because the storage itself is destroyed, or it is not secure enough if not performed every time sensitive data is deleted.
2. All the file system data can be encrypted. This protects deleted as well as non-deleted data. However, this solution has a problem of key management that is common for all encryption systems (i.e., the key safety and revocation). In addition, certain forms of encryption are not legal in some countries. Encryption is also CPU intensive, and adds overheads for many file system operations—not just erasure-related operations [28].
3. Overwriting of the whole storage media is simple. It can be performed entirely from the user mode or even assisted at the hardware level. Unfortunately, overwriting a whole file system is inconvenient, because all files are erased (even those the user wants to keep). Therefore, users are likely to refrain from this procedure.
4. On demand per-file overwriting from the user level

can be implemented using library extensions, but does not overwrite all the meta-data and does not work with statically linked binaries.

5. Driver-level and firmware-level overwriters cannot reliably detect delete and truncate operations and cannot support per-file erasure policies [24].
6. File systems can reliably detect delete and truncate operations and perform per-file on-demand overwriting. It is convenient because the data purging is performed automatically as needed. This method is secure because all the necessary file system operations are intercepted. Unfortunately, this method requires modification of every file system that is used to store sensitive information.

We have designed a system for automatic file system instrumentation. Our system can automatically add functionality to existing file systems if the file systems' source code is available. The instrumentation script parses target file system code and file system extensions written in the FiST language [31], then it automatically updates the file-system's source code with the new functionality. The FiST language was originally designed for stackable file system development and describes file system extensions in an OS-independent manner. Our instrumentation system uses FiST in a more general way, because it allows us to add new features not only to stackable file systems but also to any existing and *future* file system that follows the required Linux file system API.

We have designed a FiST file system extension we call *Purgefs*. Our instrumentation script generates a new file system using the Purgefs extension code and the code of target file system as input. Many popular file systems, especially those used for mobile storage, map the same file portions to the same storage locations. When instrumented with Purgefs, such file systems can overwrite files' data and meta-data upon file delete and truncate operations. Overwrite policies can be configured based on file names, file attributes, or directory attributes. Therefore, Purgefs overwrites data only as needed, providing security and convenience with only a minimal performance degradation. Because FiST is portable across OSs and file systems, Purgefs can be applied to most of the existing and future file systems automatically. In particular, we have applied it to Ext2 [4], vfat, msdosfs, NFS [17], and ramfs. If a file system's source code is not available, then Purgefs can be applied to a null-layer pass-through stackable file system [30]. The stackable file system can, in turn, be mounted over lower file systems and provide the same Purgefs functionality to the users, with an additional small overhead.

Purgefs supports policies to trade-off performance for the level of purging guarantee. For example, multiple synchronous overwrites can add significant overhead but are more secure; conversely, a single asynchronous overwrite is less secure but adds much smaller overheads. We have experimentally demonstrated that in the common case of a single synchronous overwrite policy and a CPU-bound workload, Purgefs overheads are undetectable.

The rest of this paper is organized as follows. We describe prior work in Section 2. Section 3 describes Purgefs's design. Section 4 describes our implementation. We evaluate our system in Section 5. We conclude in Section 6.

2 Background

Deleted data can be easily found on a disk via the `/dev` interface if the deleted file is small and a portion of the file is known. However, it is necessary to know either the file structure or the file system structure to recover large files with unknown data. For example, the Free Downloads Center [6] offers dozens of programs to recover deleted files from popular file systems.

It is reported that overwritten data on magnetic media can potentially be recovered using Magnetic Force Microscopes (MFM) [8]. However, such a method is expensive because it requires microscopes that can scan large surfaces. A more promising approach is to use a spin-stand to collect several concentric and radial magnetic surface images. Later, these images can be processed to generate a single surface image [14]. However, it is unclear if such techniques are feasible. Generally, it is believed that multiple overwriting of the data can make the recovery of data much more difficult [10].

There are several ways to prevent the recovery of erased data. The National Institute of Standards and Technology (NIST) recommends that magnetic media be degaussed or overwritten at least three times [9]. The National Industrial Security Program Operating Manual (NISPOM) describes several methods for clearing or sanitizing sensitive information [23]. Table 1 lists the software-based methods. Different methods are applicable to different types of storage. For example, method *f* is designed for SRAM-based storages. We do not include here other NISPOM methods which are hardware-related as they are not relevant to this paper.

Existing systems to erase the data reliably from the storage can be roughly divided into six categories:

1. Hardware devices can physically destroy or wipe out the storage. For example, hardware degaussers are manufactured by several companies. Such degaussers generate strong magnetic fields that not only wipe out magnetic information but can also physically destroy the storage media. Strong magnetic fields can bend hard drive platters and make their rotation impossible [19].
2. Some storage devices, virtual memory subsystems, and file systems support encryption [15, 20, 21, 29]. For example, NCryptfs is a stackable file system that

ID	Clearing/sanitizing method
c	Overwrite all addressable locations with a single character.
d	Overwrite all addressable locations with a character, its complement, then a random character and verify.
e	Overwrite all addressable locations with a character, its complement, then a random character.
f	Each overwrite must reside in memory for a period longer than the classified data resided.
h	Overwrite all locations with a random pattern, all locations with binary zeros, and then with binary ones.

Table 1: Software-based methods of clearing or sanitizing storage defined in the National Industrial Security Program Operating Manual (NISPOM).

encrypts files, directories, or whole file systems with individual keys [29]. However, the security of such encryption systems is not much better than the security of the root account. Once the privileged account is compromised, the encryption keys can be easily sniffed and used to decrypt both deleted and non-deleted data. Also, strong encryption is illegal in some countries and adds non-negligible overheads.

3. Whole partition overwriters are simple to implement (a one line shell script can overwrite an entire file system). New ATA drives even support a special mode for overwriting a whole drive (this feature is optional for SCSI drives). This can overwrite all tracks including bad and remapped ones. Also, hard drive assisted erasure is much faster. Common hardware erasure times are typically 20 or more times faster on modern hard drives than user-mode overwriters [11]. Nevertheless, whole file system overwriters are only suitable for sanitizing a drive before its disposal. It is inconvenient to overwrite a whole drive just to delete a single file.
4. There are a number of user mode tools for on-demand data overwriting [5, 16, 26]. Some of them substitute utilities like *rm* with wrappers that overwrite the data prior to deleting a file. Other tools instrument common libraries to intercept truncate and delete operations. Generally, these tools are not secure because they cannot intercept all the programs or operations that can leave file data on the disk.
5. Driver-level and firmware-level overwriters can properly locate data on the lower-level storage even for Journaling and Log-structured file systems. Unfortunately, low-level erasers cannot reliably detect erasure and truncate file system operations [24]. Also, they cannot support per-file erasure policies.
6. The Ext2, Ext3, and Reiserfs file systems, among others, support a special per-file attribute to mark files or directories that contain sensitive information and require secure deletion. Thus, the following command can mark a file as sensitive: `chattr +s` *`filename`*. Unfortunately, the attribute is not currently used and is maintained for future use. Several authors proposed patches that add purging functionality to particular file systems. For example, Bauer et. al. modified Ext2 to erase the data and meta-data on unlink and truncate operations [3].

Incremental addition of OS features and control interception points is a well developed research area. We will focus on the four methods most relevant to file systems:

1. A file system source code can be modified directly. For example, data overwriting functionality was manually added to the Ext2 file system [3]. This method has a clear drawback: new code is required not only for every OS and every file system but also for different versions of OSs and file systems.
2. Modern OSs provide hooks that allow dynamic instrumentation. For example, DTrace [2] on Solaris and Linux Security Modules [27] on Linux provide interception points in many places. However, these instrumentation APIs are not portable across OSs and do not intercept all file system operations. For example, memory-mapped operations are not intercepted.
3. Some of the file system operations may be intercepted and changed entirely from the user-mode. First, system utilities can be substituted with wrapper scripts or other binaries. Second, system libraries can be instrumented directly. In both cases, some of the programs will not be instrumented either because they are not replaced or because they are statically linked. Moreover, some file system operations cannot be changed this way; this includes popular memory-mapped operations.
4. Stackable file systems are portable across OSs and across file systems [31]. They can be mounted over any lower file system, several file systems, or only a single directory or file. However, stackable file systems add overheads for all file system operations even if only a single operation is modified. In addition, stackable file systems use twice as many Virtual File System objects thus reducing the overall size of file system caches.

3 Design

Automatically erasing data when files are truncated or removed balances security and convenience. Such erasure must be performed by the file systems because not all operations can be intercepted. Unfortunately, this requires modification of every file system that may be used to store sensitive data. We have designed Purgefs: a file system extension which can be automatically applied to many file systems running on a number of OSs.

Many file systems map the same portion of a file to a fixed location on the disk. Therefore, overwriting the file data at the file system level results in overwriting of the data on the physical storage. Purgefs overwrites file data and meta-data one or more times using a number of data patterns for overwriting. For example, it can support all of the relevant methods shown in Table 1.

Purgefs can overwrite data synchronously or asynchronously. In a simple synchronous mode, Purgefs blocks until every overwrite operation completes. In asynchronous mode, Purgefs can remap data pages to a temporary file and overwrite them later on using a kernel thread. By remapping only the appropriate data pages, Purgefs can asynchronously overwrite only a portion of a file upon truncate operations. Overwriting asynchronously is simple and also improves performance, but may lower the level of purging guarantee because the file may not be completely overwritten if the system crashes (e.g., because of a power failure).

Overwriting decisions can be customized to better balance security with performance. Purgefs can overwrite all files or only files marked with a special attribute using the `chattr +s` *`filename`* command. The number of overwrites and the type of overwrite patterns can be configured as mount-time options. A single overwrite with any data can prevent recovery in most cases. Multiple overwrites make recovery using advanced methods even less probable.

3.1 Automatic Purgefs Addition

Stackable file systems can incrementally add functionality to the lower file systems. Figure 1 shows a *Base0fs* stackable file system that passes through all the file system operations from the *Virtual File System* (VFS) to the lower file system. Unfortunately, stackable file systems

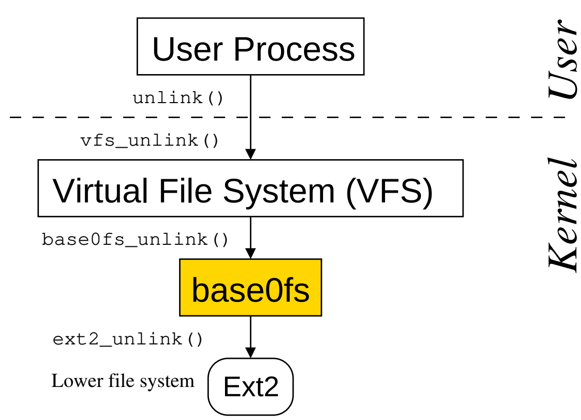

Figure 1: Base0fs stackable file system mounted over Ext2.

add overheads to all file system operations. Direct file system source code instrumentation produces file systems that run more efficiently, because only the necessary operations are instrumented and the compiler has the flexibility to optimize the code. Unfortunately, such instrumentation requires manual work for every file system and every OS version. We decided to combine the benefits of both approaches. We have created a script that automatically instruments a subset of file system operations directly in the source code. If a file system source code is unavailable, then the same script can instrument the Base0fs file system. Base0fs can then be mounted over a file system whose source code is not available, adding some overhead but maintaining the same purging functionality.

Figure 2 shows that the instrumentation script processes both the target file system and the Purgefs extension. Based on the information contained in both, Purgefs generates a new instrumented file system.

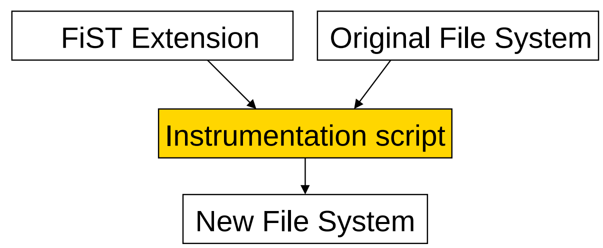

Figure 2: Instrumentation script operation.

We have designed our instrumentation script to be flexible. In particular, it supports file system extensions written in the FiST language. FiST files have a structure similar to the structure of the YACC file format [12]. Every FiST file consists of three sections separated with a `%%` line. The first section contains code and macros added to a generated header file. The middle section describes operations that require instrumentation. The last section describes routines for a separate generated source file. FiST is a C-based language. Because popular OSs are written in C, this allows direct insertion of C code from FiST files into the appropriate locations of file system code.

```
%{
   /* print out return values */
%}
%%
%op:all:postcall {
  printk("OP_%s: %d (pid: %d)\n",
         %op, fistLastErr(), %pid);
}
%%
/* no extra routines are needed */
```

Figure 3: FiST file system extension that prints out the return values of every file system operation.

```
static int ext2_writepage(struct page *page, struct writeback_control *wbc)
{
   return block_write_full_page(page, ext2_get_block, wbc);
}

static int ext2_writepage(struct page *page, struct writeback_control *wbc)
{
  {
    int fist_tmp_ret_val = block_write_full_page(page, ext2_get_block, wbc);
    printk("OP_%s: %d (pid: %d)\n",
           "writepage", fist_tmp_ret_val, current->pid);
    return fist_tmp_ret_val;
  }
}
```

Figure 4: Original (top) and generated (bottom) writepage operation of the Ext2 file system.

Figure 3 shows an example FiST file system extension intended for file system debugging. It adds code that prints out return values of every file system operation. Let us consider the actions performed by the instrumentation script to generate the target file system on Linux for FiST extension shown in Figure 3:

1. The script replaces `printk`, and `%pid` with the strings appropriate for the target OS. for Linux, `printk` is kept intact but is replaced with `printf` for FreeBSD.
2. The script looks for file system operation vectors declared in the source code.
3. The script creates wrapper functions for operations that use the generic OS methods.
4. The script adds the requested `printk` statement before every `return` statement of all operations.
5. The script substitutes `%op` and `fistLastErr` with operation-specific strings: the operation name and the variable name that contain the last error. If necessary, the return value is converted to the integer form. For example, the `PTR_ERR()` macro is used under Linux if the previous function returned a pointer.

Figure 4 shows an original and the generated `writepage` operations code for an Ext2 file system. We can see that the instrumentation script created a temporary variable to store and report the internal return value.

The instrumentation script converts abstract *vnode* objects into objects used by target OSs and refers to file properties via a unified vnode object. For example, the script understands that every vnode under Linux is represented by several objects: file, dentry, and inode. Thus, the script uses the dentry object to access the file name, and the inode object to access the file size. At the same time, developers do not need to know about it and can assume that these are the abstract vnode properties.

The instrumentation script design allows file system developers to concentrate on their new concepts or features instead of the implementation for every file system and OS. These actions are done by a single and specific instrumentation script. For example, currently we are investigating a general way to overwrite data on journaling file systems. When completed, we will not need to modify the instrumentation script, but only modify the FiST extension. Even better, if some OS property changes, we will not even need to modify the FiST extension.

4 Implementation

Our current Purgefs prototype consists of only 84 lines of FiST code. Purgefs synchronously overwrites during the first $N-1$ times (if $N>1$) and asynchronously overwrites during the last overwrite. This implementation allows our current Purgefs prototype to remain simple because one asynchronous overwrite can be performed using existing OS functions. This is important because it allows us to apply Purgefs using the current instrumentation script implementation. Fortunately, this lets us perform the overwriting completely asynchronously in the most common case of a single overwrite.

Purgefs supports single and triple overwriting modes. In particular, it supports modes *c* and *e* from Table 1. In the single overwrite mode, it overwrites the data with zeroes. In the triple overwrite mode, it overwrites the data with 0x55, 0xAA, and then random pattern. (0x55 and 0xAA are complementary alternating bit numbers.)

The instrumentation script is written in *Perl* [25]. We believe Perl is an appropriate choice because most of the time the instrumentation process searches the code for matches of regular expressions—a scenario Perl is especially suitable for. Aside from string matching and replacing, the instrumentation system parses only small portions of the C code. For example, it determines the types of functions and their arguments. We found that a simple top-down parser is sufficient for this purpose.

Our current unified instrumentation script can instrument Linux 2.4, Linux 2.6, and FreeBSD 5.4 file systems. It consists of 305 lines of Perl code. We have successfully applied and verified the functionality of Purgefs with all the non-journaling file systems we tried. In particular, we instrumented the following six file systems:

Ext2 (Linux) which is commonly used on Linux.

vfat (Linux) and **msdosfs** (FreeBSD) which are usually used on portable FLASH drives.

ramfs (Linux) to purge files from memory (policy *h* from Table 1 is not enforced).

NFS (Linux, FreeBSD) allows instrumentation on the client side without instrumenting the server side.

Base0fs (Linux, FreeBSD) is a stackable file system which can be mounted over any other file systems whose source code is unavailable.

5 Evaluation

We evaluated our system on a P4 1.7GHz machine with 1GB of memory. Its system disk was a 30GB 7200 RPM Western Digital Caviar IDE and was formatted with Ext3. In addition, the machine was equipped with one 18.4GB Ultra320 SCSI disk formatted with Ext2. We used this separate SCSI disk for running the benchmarks.

We remounted the test file systems before every benchmark run to purge file system caches. We ran each test at least ten times and used the Student-t distribution to compute the 95% confidence intervals for the mean elapsed, system, user, and wait times. Wait time is the elapsed time less CPU time used and consists mostly of I/O, but process scheduling can also affect it. In each case, the half-widths of the confidence intervals were less than 5% of the mean. The test machine was running a Fedora Core 3 Linux distribution with a vanilla 2.6.11.7 kernel.

We evaluated and compared the following three configurations: vanilla Ext2 (EXT2); Ext2 instrumented with secure delete functionality with the overwrite-once policy (EXT2PURGE-C); and overwrite three times according to method *e* of Table 1 (EXT2PURGE-E). We chose Ext2 because it is a popular file system for manual extension with secure delete functionality. Therefore, future and past developers may conveniently compare their systems with Purgefs's overheads.

5.1 CPU-Bound Workload

First, we evaluated Purgefs under a compile workload—a CPU-intensive workload that is similar to the workloads generated during normal user activities (i.e., more CPU activity generated than file system I/O activity). We compiled the Am-utils [18] package version 6.1.1 using the EXT2, EXT2PURGE-C, and EXT2PURGE-E configurations and compared the overheads of the instrumented Ext2 with vanilla Ext2.

Am-utils contains over 60,000 lines of C code in 430 files. The build process begins by running several hundred small configuration tests to detect system features. It then builds a shared library, ten binaries, four scripts, and documentation: a total of 152 new files and 19 new directories. Although the Am-utils compile is CPU intensive, it contains a fair mix of file system operations. We used Tracefs [1] to measure the exact distribution of operations. The Am-utils build process uses 25% writes, 22% lseek operations, 20.5% reads, 10% open operations, 10% close operations, and the remaining operations (12.5%) are a mix of readdir, lookup, etc. Most importantly for us, the build process deletes 4,696 files during these phases.

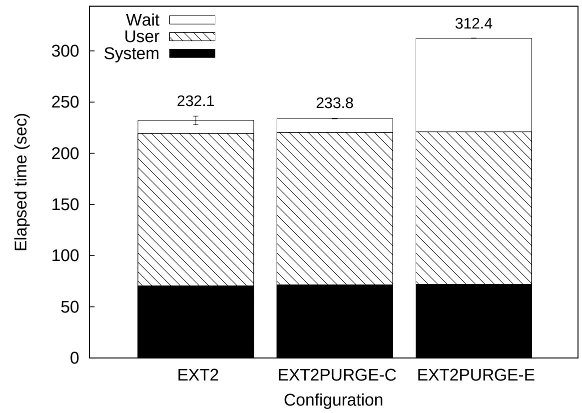

Figure 5: Am-utils build times.

Figure 5 shows the measured build times. As we can see, Purgefs in asynchronous single-overwrite mode does not add any noticeable overheads under the CPU-intensive workloads. The EXT2PURGE-E configuration has a noticeable 35% elapsed time overhead because the first two writes are performed synchronously. Because this overhead is mostly I/O, faster disks will improve Purgefs performance.

5.2 I/O-Bound Workload

We evaluated our system using an I/O-intensive workload generator. Postmark [13] simulates the operation of electronic mail servers. It performs a series of file appends, reads, creations, and deletions. We configured Postmark to create 20,000 files, between 512–1M bytes, and perform 200,000 transactions. We selected the create/delete and read/write operation ratios with equal probability.

Naturally, Purgefs adds wait time overheads under I/O-intensive workloads because the operation is I/O-bound and erasing operations compete with other I/O operations. As we can see in Figure 6, EXT2PURGE-C is 81% slower and EXT2PURGE-E is 22 times slower than the vanilla Ext2. Such high overheads are caused not only by the I/O-bound nature of the workload but also by the fact that every fourth operation is a deletion. We believe that 81% overhead even under such severe workload conditions is a promising result showing that Purgefs overheads may be acceptable in most cases.

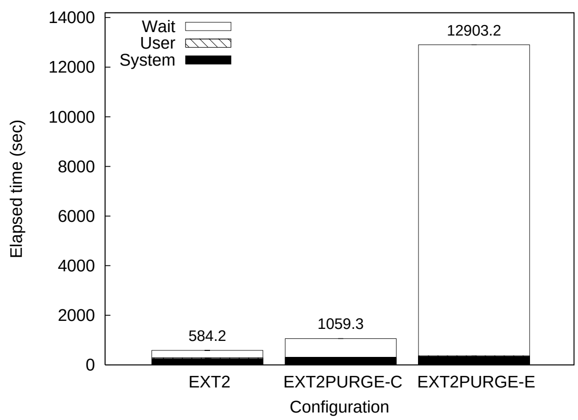

Figure 6: Postmark benchmark times.

5.3 File System Generation Time

Finally, we evaluate the efficiency of the instrumentation program. It is less important than the runtime overheads. However, short instrumentation times are more desirable for file system developers. At first glance it may seem that the process of file system generation can take a considerable amount of time. Indeed, the generation process requires at least two scans of the file system source code. Table 2 shows the times necessary to add secure deletion functionality to several popular file systems and their compilation times. Note that the overheads of compilation times were indistinguishable. As we can see, in all the cases, the instrumentation times were much smaller than the compilation times of these file systems. Therefore, the instrumentation time can be considered small.

File system	Instrumentation time (seconds)	Compilation time (seconds)
Ext2	1.2	14.6
vfat	1.1	6.1
msdosfs	1.2	5.8
NFS	1.3	16.5
ramfs	1.2	6.0
Base0fs	1.3	11.7

Table 2: Purgefs's instrumentation time for several file systems, and the original compilation times of these file systems.

6 Conclusions

Secure erasure of file data from the storage, upon file delete, is consistent with users' perception of what a delete operation should do. Unfortunately, existing file systems do not provide this guarantee.

We have designed Purgefs—a highly portable, secure, and convenient data-purging system for everyday use. Purgefs is portable because it is designed as a file system extension which can be automatically added to most existing file systems. Purgefs is secure because it performs data purging immediately after every delete and truncate operation. Purgefs is convenient because it operates transparently; and it is relatively efficient because it operates asynchronously if desired.

Purgefs can be used with any block-based and network-based file system that maps the same file portions to the same storage location. We have tried and successfully tested Purgefs with the Ext2, vfat, msdosfs, ramfs, NFS, and Base0fs file systems. Ext2, vfat, and msdosfs are frequently used for mobile storage, where secure deletion is especially critical. NFS instrumentation demonstrates that Purgefs's client-side use is sufficient for secure deletion purposes. The Base0fs file system allows Purgefs to be used with file systems whose source code is unavailable. Moreover, many other existing and *future* file systems can be instrumented without any additional development efforts.

We have demonstrated that under non-intensive workloads typical for users, Purgefs's operation is completely non-intrusive and its overheads are negligible. We have shown that even under severe I/O-intensive and delete-intensive workloads, Purgefs has reasonable overheads in configurations that sufficient for most users: the single-overwrite mode.

Future Work. We are currently investigating more sophisticated approaches which will allow us to use Purgefs with journaling and log-structured file systems. For example, we are investigating instrumentation of the device drivers in addition to file systems, and the Linux Journaling (JBD) API.

Our instrumentation system can have a broader application. In particular, it can be used to add many different features to file systems if extended to support a wider set of FiST language primitives. For example, we plan to implement an encryption file system extension.

The instrumentation system can be ported to a number of OSs (e.g., Solaris). Also, a more advanced instrumentation system will enable us to decrease the Purgefs overheads further and use more sophisticated asynchronous modes. Lastly, Purgefs can be easily extended to support more overwriting modes and patterns.

Acknowledgments

This work was partially made possible by NSF CAREER award EIA-0133589, NSF Trusted Computing Award CCR-0310493, and HP/Intel gift numbers 87128 and 88415.1. We would like to thank Charles P. Wright for his help with the paper's preparation.

References

[1] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 129–143, San Francisco, CA, March/April 2004. USENIX Association.

[2] B. Cantrill and M. W. Shapiro and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 15–28, 2004.

[3] S. Bauer and N. B. Priyantha. Secure Data Deletion for Linux File Systems. In *Proceedings of the 10th Usenix Security Symposium*, pages 153–164, Washington, DC, August 2001. USENIX Association.

[4] R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings to the First Dutch International Symposium on Linux*, Seattle, WA, December 1994.

[5] *CyberScrub Secure File Deletion / Internet Privacy Utility*. `www.cyberscrub.com`.

[6] Free Downloads Center. Linux Disk DoD. `www.freedownloadscenter.com/Best/linux-disk-dod.html`.

[7] S. Garfinkel and A. Shelat. Remembrance of Data Passed: A Study of Disk Sanitization Practices. *IEEE Security and Privacy*, 1(1):17–27, January 2003.

[8] R. Gomez, A. Adly, I. Mayergoyz, and E. Burke. Magnetic Force Scanning Tunnelling Microscope Imaging of Overwritten Data. *IEEE Transactions on Magnetics*, 28(5):3141–3143, September 1992.

[9] T. Grance, M. Stevens, and M. Myers. *Guide to Selecting Information Security Products*, chapter 5.9: Media Sanitizing. National Institute of Standards and Technology (NIST), October 2003.

[10] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *Proceedings of the Sixth USENIX UNIX Security Symposium*, pages 77–90, San Jose, CA, July 1996. USENIX Association.

[11] G. Hughes. CMRR Protocols for Disk Drive Secure Erase. Technical report, Center for Magnetic Recording Research, University of California, San Diego, October 2004. `http://cmrr.ucsd.edu/Hughes/CmrrSecureEraseProtocols.pdf`.

[12] S. C. Johnson. Yacc – Yet Another Compiler-Compiler. Technical Report CS-TR-32, Bell Laboratories, Murray Hill, NJ, July 1975.

[13] J. Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[14] I. Mayergoyz, C. Seprico, C. Krafft, and C. Tse. Magnetic Imaging on a Spin-Stand. *Journal of Applied Physics*, 87(9):6824–6826, May 2000.

[15] Microsoft Research. Encrypting File System for Windows 2000. Technical report, Microsoft Corporation, July 1999. `www.microsoft.com/windows2000/techinfo/howitworks/security/encrypt.asp`.

[16] *Overwrite, Secure Deletion Software*. `www.kyuzz.org/antirez/overwrite`.

[17] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3 design and implementation. In *Proceedings of the Summer USENIX Technical Conference*, pages 137–152, Boston, MA, June 1994.

[18] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.1b3 edition, July 2003. `www.am-utils.org`.

[19] Peripheral Manufacturing Inc. Degaussing (Erasing) Equipment. `www.periphman.com/degaussing/tape-degaussing/`.

[20] M. Petullo. Implementing encrypted home directories. *Linux Journal*, pages 62–68, August 2003.

[21] N. Provos. Encrypting virtual memory. In *Proceedings of the Ninth USENIX Security Symposium*, Denver, CO, August 2000.

[22] J. Rosenbaum. In Defence of the DELETE Key. *The Green Bag*, 3(4), Summer 2000. `www.greenbag.org/rosenbaum_deletekey.pdf`.

[23] Defense Security Service. *National Industrial Security Program Operating Manual (NISPOM)*, chapter 8: Automated Information System Security. U.S. Government Printing Office, January 1995.

[24] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or Death at Block-Level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 379–394, San Francisco, CA, December 2004. ACM SIGOPS.

[25] L. Wall, H. Stenn, and R. Manfredi. dist-3.0. Technical report, Comprehensive Perl Archive Network (CPAN), 1997. `ftp.funet.fi/pub/languages/perl/CPAN/authors/id/RAM`.

[26] *Wipe: Secure File Deletion*. `http://wipe.sourceforge.net/`.

[27] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, August 2002.

[28] C. P. Wright, J. Dave, and E. Zadok. Cryptographic File Systems Performance: What You Don't Know Can Hurt You. In *Proceedings of the 2003 IEEE Security In Storage Workshop (SISW 2003)*, pages 47–61, Washington, DC, October 2003.

[29] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, San Antonio, TX, June 2003. USENIX Association.

[30] E. Zadok. The FiST home page. `www.filesystems.org`, 1999.

[31] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proc. of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000. USENIX Association.