# Ransomware-proof: A Data-Poisoning Resistant Filesystem

Alistair Mann

al@pectw.net

http://www.dprfs.com

**Abstract:** Accidental data loss has always been with us, and its fraternal problem of malicious loss has got worse in recent years with the advent of ransomware. This paper offers a mitigation to both through abstracting out filesystem changes from the filesystem itself, offers technical means to comply with data-protection laws that mitigation makes more relevant, and presents a prototype implementation suitable for workgroup environments.

## *Introduction*

Data-poisoning occurs when a file is stored or communicated without the user or owner being aware that that file has been:

•corrupted (e.g., overwriting accidentally selected text)

•lost (e.g., saving a spreadsheet as a .csv quietly loses worksheets other than that currently displayed)

•encrypted (e.g., ransomware, lost passwords to archive files)

•deleted (e.g., rogue employee[1], rogue merchant[2 3])

Ransomware is a relative novelty of the encrypted type where the password is at no point known to the victim, but would be released for a payment: an extortion. As malefactors improve their threat[4], and that knowledge reaches others[5], handling the general problem of data-poisoning would reduce the personal loss of photos and videos[6], the economic loss with companies having to shut down temporarily[7] or even permanently[8], the moral loss[9] of Police Departments paying the extorter to

[1] “Former Gray Wireline Employee Admits to Computer Intrusion“ https://www.fbi.gov/elpaso/press-releases/2010/ep120210.htm

[2] “Amazon Erases Orwell Books From Kindle” http://www.nytimes.com/2009/07/18/technology/companies/18amazon.html?_r=0

[3] “Apple Stole My Music“ https://blog.vellumatlanta.com/2016/05/04/apple-stole-my-music-no-seriously/

[4] “TeslaCrypt 2.0 ransomware comes with significant improvements“ http://securityaffairs.co/wordpress/38616/cyber-crime/teslacrypt-2-0-ransomware.html

[5] “If the victim pays up, the original author gets a cut … and the rest goes to the "script kiddie" who deployed the attack.” http://uk.businessinsider.com/ransomware-as-a-service-is-the-next-big-cyber-crime-2015-12

[6] “my friend’s pictures were somehow damaged.” https://askleo.com/another-story-of-data-loss-and-what-could-have-been/

[7] http://www.dailyrecord.co.uk/news/scottish-news/ransomware-virus-causes-closure-national-7658693#ijreOjIj0rxfelIP.97

[8] “every employee had access to … 82 percent of the files. Effectively, one infection would put the firm out of business.” http://www.itbusinessedge.com/blogs/unfiltered-opinion/varonis-frightening-report-reveals-companies-are-easy-prey-for-locky-ransomware.html

[9] FBI: “we often advise people just to pay the ransom.”, https://securityledger.com/2015/10/fbis-advice-on-cryptolocker-just-pay-the-ransom/

access the department's own files[10], and the societal loss from these threats to data[11].

Data-poisoning is possible because of two design choices that might now be viewed as flaws. First, older filesystems provide no native or effective “undo” facility. While there are options[12], what's available afterwards can only be a reconstruction, often only partial. If an attack can't be avoided, it's preferable – and with a data-poisoning resistant filesystem (dprfs), obtainable – to have the original filesystem state remain available.

A second, even older choice was that no facility may be denied to a superuser. While the principle of least privilege mandates that there are superusers (or at least super*usage*[13]), there's no historical requirement that a particular facility be made available to the superuser just because it can be. Rather, it's preferable to recognise the superuser on a running system can have different privileges to the superuser of an offline system, even though the operating system is the same.

I suggest the safety of data committed to a filesystem should now be viewed as a responsibility of that filesystem, and the choices above compromise that responsibility. The two major filesystems in use, NTFS under Windows and HFS+ under MacOS, were developed in times of relative constraint. In late 2015 a hard drive can be cheaply found with a 3Tb capacity[14]. HFS+ was introduced in 1998[15] when drives had 300 times less capacity[16]. Likewise NTFS was introduced in 1993[17] when drives had *over 9000* times less capacity[18]. Maximising data capacity over data safety – among other decisions – led to a design choice whereby user action on the filesystem acts directly on the disk. This is contrary to user expectation: for example deleting wordprocessed text doesn't irrevocably remove that text – one can usually “undo” it back into place. That the implementation of these filesystems have not changed in so long makes them mature in computing terms and this rightly militates against major change – which may help explain why new filesystems have not been tried[19].

Until such time that filesystems arrive already resistant to data-poisoning, a prototype can prove much of that utility if it reserves some functions to an offline user (eg, “final” deletion requires physical access), reimplements the filesystem facilities to be poisoning resistant, abstracts user experience of those facilities to match their expectations, and implements technical means for complying with data laws. As this paper documents the technical issues involved, it also documents the creation of a data-poisoning resistant filesystem on a NAS device suitable for live use.

[10] The department paid $750 for two Bitcoins ... to decrypt several images and word documents in its computer system” http://www.heraldnews.com/article/20131115/News/311159409

[11] “MedStar Health turns away patients after likely ransomware cyberattack“ https://www.washingtonpost.com/local/medstar-health-turns-away-patients-one-day-after-cyberattack-on-its-computers/2016/03/29/252626ae-f5bc-11e5-a3ce-f06b5ba21f33_story.html

[12] There are two forms of recovery after data loss. First, pre-existing tactics such as in-situ backups with Windows' Shadow Copies, MacOS' File Versions and 3rd party backup software. Second, undertaken afterwords: data-recovery of deleted files offers some relief, as does re-downloading email received and sent; Some files can be retrieved from social media, others can be obtained from original purchase and so forth.

[13] “Plan 9 has no super-user.” http://doc.cat-v.org/plan_9/4th_edition/papers/9

[14] 2015-12-18, 3Tb for $84.99 at Newegg, http://www.jcmit.com/diskprice.htm

[15] HFS Plus was introduced with Mac OS 8.1” https://developer.apple.com/legacy/library/technotes/tn/tn1150.html#HFSPlusBasics; (Mac OS 8.1 was) “Released on 1998” https://en.wikipedia.org/wiki/Mac_OS_8#Mac_OS_8.1 (Uncited)

[16] 4Gb to 16.8Gb hard drives available in 1998, http://edition.cnn.com/TECH/computing/9901/21/honkin.idg/

[17] https://en.wikipedia.org/wiki/NTFS#cite_note-Custer.2C_Helen-7

[18] “ … we take for granted that typical hard drives have capacities from 100MB to 300MB or more” PC Mag, April 13 1993, p343, Vol. 12, No. 7, ISSN 0888-8507

[19] Although see the abandoned WinFS: “Why WinFS had to vanish“ http://www.theguardian.com/technology/2006/jun/29/insideit.guardianweeklytechnologysection and the ReFS (“Resiliant File System) https://blogs.technet.microsoft.com/askpfeplat/2013/01/01/windows-server-2012-does-refs-replace-ntfs-when-should-i-use-it/ neither of which offered data-poisoning resistance.

## *Overview*

This paper first looks at differences between a classical filesystem and a data-poisoning resistant filesystem in terms of concepts, implementations and extensions.

We then look at how files are themselves used to accomplish work – not every file is born equal.

Next at the prototype

We look at the implications beyond the filesystem of adopting a dprfs.

## *Features of the data-poisoning resistant filesystem (dprfs)*

Some kind of intro to the difference between a dprfs and classical system

### Delete doesn't mean remove, save doesn't mean overwrite

In a classical filesystem (ext4, ntfs, hfs+ etc), to remove() a filesystem object is to directly detach it from all indicies capable of addressing it and so potentially lose that data for good[20]. With dprfs, unlink() & rmdir() cause the object to be *flagged* as deleted, so maintaining both the object, and its deleted status.

Similarly, to save in a classical filesystem is to replace the original filesystem object with a new, updated copy, again potentially losing the original[21]. With drpfs, a save is made *in addition* to the original, so maintaining access to both.

### Beyond Use

With delete modified as above, the potential arises that the system falls afoul of external issues[22] with data being deliberately persisted. This paper proposes to extend the filesystem interface with a new call, put_beyond_use(), which takes a date/time (either absolute or relative) as argument. On successful return, the object so handled will not be accessed by any compliant call where the date/time has been exceeded.

In practical terms, a merely deleted file or directory can be recovered by an authorised user of that running system in a manner similar to the MS-DOS "undelete"[23] command from the 1990s; but a file or directory put beyond use must have the volume physically removed before access can be gained.

### Restricted manipulations

During the normal use of the operating system, no user – specifically including any user with superuser privileges – may manipulate files in the following ways:

•Deletion. No unlinking anything from the filesystem

•Touching. No date changing of an existing link. Although it would be acceptable to create a new object with the required change

•Beyond use. No access if a link has been put "beyond use"

[20] "deletes a name from the file system …  the space it was using is made available for reuse." http://linux.die.net/man/3/remove

[21] "After a *write*() to a regular file has successfully returned: … Any successful *read*() from each byte position in the file that was modified by that write shall return the data specified by the *write*() for that position" http://linux.die.net/man/3/write

[22] "While this has more to do with litigation exposure, compliance and discovery, 2.8 million files had been untouched for six months or longer." http://www.itbusinessedge.com/blogs/unfiltered-opinion/varonis-frightening-report-reveals-companies-are-easy-prey-for-locky-ransomware.html

[23] "Microsoft DOS undelete command" http://www.computerhope.com/undelete.htm

This must be enforced by the relevant driver having no code compiled in to perform the first and second, and specifically implementing the third. It is a security risk and so incompliant with dprfs to have a driver that can perform these functions, but decide not to, at some runtime check.

## Offline user to whom restricted manipulations are reserved

If a restricted manipulation is required for whatever reason, the only user capable of it is a super*er*user capable of restarting that device from external media, or physically removing the hard drive, etc. In this way, the restricted manipulations are reserved to a user with known physical access (or remote access via iDrac[24], etc.)

This is implemented by allowing compiled-in support for the manipulations where that driver is to be compiled-in to an offline operating system. “Offline” is here used in the sense the device is in a maintenance mode where processing for ordinary users is suspended, and the device will not process remote connections.

That these manipulations can only be accomplished offline complies with laws such as “protect that information from unauthorized access, destruction, use, modification, or disclosure” in California[25] and “technical measures shall be taken against unauthorised processing of personal data and against destruction of, or damage to, personal data” in the UK[26] as the decision to override those policy objectives is put more in the hands of a physical user and away from remote users exploiting exposed programming.

## The linkedlist as dprfs's fundamental unit

A classical filesystem sees one addressable location for each file or directory: there can be only one object addressed “/path/to/file”. Change it, and the original is gone; delete it, and the original is no more. With dprfs, we add a new dimension whereby we maintain a record of how the object was at a particular point: when a file is changed, we record it at that new point; when a directory is deleted, we add a flag indicating the existing directory is deleted. This is accomplished by storing filesystem objects as a *linkedlist*[27]: The head of every linkedlist is considered the current version of the file persisted by the list, and is the default used for access when processing.

As an edit no longer overwrites the original, that original remains available on disk should it be needed. As the edit points to the previous, then the system can easily work back through the links to a particular version. As a one-way linkedlist (that is, originals do not point to a later edit), that linked list can be reverted to an earlier point simply by unlinking or skipping those later points in turn. And by maintaining separation between edit and original, the head can be made read/write, and the originals read-only.

The user's expectation of a single file is maintained when normal action on the linkedlist can only act on the head.

In the linkedlist paradigm, the poisoning of a file merely causes a new copy of that file to be poisoned – the original remains inviolate.

## Each link contains metadata, resource fork and payload

Classical metadata – such as ownership info, permissions, hidden status and so forth – apply to the linkedlist rather than the individual link.

[24] “integrated Dell Remote Access Controller” http://en.community.dell.com/techcenter/systems-management/w/wiki/3204.dell-remote-access-controller-drac-idrac

[25] "Student Online Personal Information Protection Act", https://leginfo.legislature.ca.gov/faces/billNavClient.xhtml?bill_id=201320140SB1177 22.2/2584/d/1

[26] “Data Protection Act 1998” 1/1/7, http://www.legislation.gov.uk/ukpga/1998/29/schedule/1/part/I/paragraph/7

[27] http://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked%20Lists/linked%20lists.html

Each link gets its own new metadata area used to store:

1.'beyond-use-on' field containing a timestamp on or after which the online driver will refuse to access data, so fulfilling data-protection expiry obligations

2.a 'deleted' flag to indicate the running system should treat the payload as deleted even though it is present on disk

3.a 'not-via' field indicating a path and filename through which this linkedlist may not be accessed. This is used to prevent a renamed file from remaining available via its original name

4.a 'payload-loc' field indicating that the payload to which this link refers can actually be found at the alternate given path. As example, this allows that a renamed file does not require its payload to be duplicated

5.a 'supercedes' field indicating which previous link this link follows.

6. Additional fields such as sha256

7.Extensions #1: the prototype treats metadata entries as key[= value] pairs, however it could be extended to handle JSON instead. [ ] so I need to put down a metadata marker indicating keyvalue vs json

8.Extensions #2: the metadata paradigm could be extended to hold further freeform fields such as "source": an array holding identifying data known about the user who uploaded the file such as IP address, smbclient name etc.
12. The payload for each link, containing a spreadsheet, archive file, email, music file etc, is the direct equivalent of the file as the user understands it.

The link paradigm could be extended to support one or more optional resource forks or alternate data streams as used by OSX or Windows.

## Forensic journal

In order to track down the exact moment a file was damaged a forensic log should also be kept. This would be a log of timestamps and changes made in the filesystem. For example, cryptolocking is often accompanied by a rename such that the malware can identify what was previously encrypted. With writes and renames in the forensic log, identifying the first moment of encryption (and thus the moment before which all might be considered uncorrupted) becomes trivial.

As the purpose of the log file is to establish what happened to a particular file, when, the log entries will need a timestamp and the filename operated on. Such an operation might occur with or without an existing file descriptor: rename(), utime(), access(), unlink() are examples of operations not needing a file descriptor; creat(), open(), close() and write() examples of operations that do.

The prototype stores entries in the form:
timestamp commiting_action: file [action_note … action_note]
where action_note is formed of
verb: <number times called>

Operations not needing a file descriptor will normally cause a single entry:

```
20160404182141897506 utime: /construction.txt.part
20160404182141899985 unlink: /construction.txt
```

Certain operations will see more than one entry:

```
20160404221115634360 rename: /cleanyourpaws.jpg
20160404221115634410 rename: /clean_your_paws.jpg
20160404221115634424 rename: /cleanyourpaws.jpg -> /clean_your_paws.jpg
20160404221115636721 lookit:  /clean_your_paws.jpg
```

Operations that do need a file descriptor could cause a great many identical entries which can be got onto a single line.

```
20160404182408980065 lookit:  /cleanyourpaws.jpg
20160404182409100218 close: /cleanyourpaws.jpg: write: 1216 creat: 1 flush: 1
```

“lookit” here is an indication from the host that it knows a file is about to be accessed in a way that will cause a destructive change, and the host is logging that fact before those changes are made. Any subsequent failure at least leaves a starting point for investigation. Changes that cannot cause a change are not logged.

Extension: Such a log could also log a compression ratio: “this file, if compressed, would be x% smaller”. This dirty measure of randomness (a marker for encryption), along with timing data, could also point to any malign changes. This would be particularly relevant in combination with the existing compression ratio: it is rare for a file to be created in the clear and then be compressed: the nearest may be *nix system logs.

## *Files are not just files*

Designing resistance for modern usage requires we understand how files are used.

### Classical files

A file is opened, written to sequentially from first byte to last, and closed. At some later point it's opened, read from and closed, perhaps deleted.

This is the simplest form to protect: at the first destructive change, the head of the linkedlist is copied forward and operations act on that copy until the next close().

Example: Windows notepad, KDE's Kate.

### Hanoi files

Named after the game[28]. Three classical files: the user believes he's only using file A, however during a save the data is saved to file B. Once complete, the unchanged file A is renamed to file C, and file B is renamed to file A. This minimises the period during which a failure would leave the filesystem is in an uncertain state. File A is different as a history shows it has always come from somewhere else; File B is different as it lives temporarily between upload and rename into place; File C also lives temporarily, until the next time there's a save.

File B and C here can be diverted to a non-dprfs filesystem as we don't want to maintain them. File A does live on a dprfs filesystem, and behaves identically to a classical file in that at the first destructive change – which will be rename() – a new head is copied into place and update.

Examples: Microsoft Word for Windows, Apple Textedit

### Datastore files

Datastore files. Here the file is treated much like random-access memory: it has to be created to a certain size first however once done, reads and writes can happen anywhere within the file, in any sequence. This has the advantage of faster access (as only changes need transferring) at an increased

[28] After the puzzle game “Tower of Hanoi” https://www.mathsisfun.com/games/towerofhanoi.html

risk of corruption.

This is the most difficult kind of file to protect as the file is always open: there's a big risk of corruption.

The solution is to journalise entries to that memory as they're made, such that replaying the journal can reconstruct to any point. The prototype as it stands doesn't implement a journal, instead specifically saving at close()

Examples: Microsoft Access for Windows, [possibly] Adobe Photoshop

### Echo files

These create a library of files through time under the control of the OS (Apple File Versions[29], Windows Shadow copy[30]), the program itself (Word autosave/autorecover[31], Emacs[32]) or 3rd party facility (git[33], Dropbox[34]). They share the objective of trying to give the user access to previous revisions. In none of these is data-poisoning resistance designed to handle malice: the previous versions by design have ownership and permissions allowing users and superusers access.

These kinds of files can be protected in the same way as classical files, however a decision should be made about whether to keep these files in a dprfs filesystem (which is already recording them) or in a non-dprfs filesystem.

## *Prototype*

### Setup

Server system unit: 2008 Dell Inspiron 1525

SSD: 1 x 120Gb Kingson SV300S37A120G

HDD: 1 x 160Gb Fujitsu MHW2160BJ[35] (Note: 7200rpm)

OS: Mint Linux 17.3 with all updates

### Performance

In the author's own tests, the prototype saves 50% slower than normal when the dprfs is backed by an SSD, and 200% to 500% slower than normal when backed by an HDD.

Both these tests are inside the range of human comfort when used on the fly (file | save as … to a dprfs network share): 500ms vs 100ms doesn't get noticed; however both are very noticeable for batch copying, such as when conducting a backup: 50mins vs 10mins  is most definitely noticeable.

Reading is X% slower.

## *Differences of behaviour between prototype and full product*

•“Beyond use” does not currently prevent an authorised user from obtaining access to so treated

[29] “How to Browse & Revert to Prior Versions of a File in Mac OS X“ http://osxdaily.com/2015/06/16/revert-to-prior-version-file-mac-os-x/

[30] “Rapid Recovery with the Volume Shadow Copy Service” https://technet.microsoft.com/en-us/magazine/2006.01.rapidrecovery.aspx

[31] “Automatically save and recover Office files“ https://support.office.com/en-us/article/Automatically-save-and-recover-Office-files-5baa2030-9768-4c6c-8d2a-1e10a8d741b1

[32] “Recovering Data from Auto-Saves“ https://www.gnu.org/software/emacs/manual/html_node/emacs/Recover.html

[33] “Getting Started - About Version Control“ https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

[34] “How do I recover previous versions of files?“ https://www.dropbox.com/en/help/11

[35] “Fujitsu MHW2160BJ: 160 GB, 7,200 RPM” http://www.tomshardware.com/reviews/speed-capacity,1659-2.html

files and directories via the rdrive. [I expect this to change shortly, with a process by which so treated files cause a cascade of changes to all previous files making them chmod 0, even to the rdrive]

## *Implications*

## Filesystem Hierarchy Standard

With a recognition that some data needs greater safeguards than others, and that two filesystems can greater balance safety and speed than one, comes recognition that the existing Filesystem Hierarchy Standard[36] needs expansion.

On *nix I would envisage something like:

| | |
|---|---|
| /home/<profile> | DPRFS: pictures, music, emails, ssh keys, “Dot files” that cannot or should not be separately and automatically rebuilt (.git, .ssh, .emacs), etc |
| /profiledata/<profile><br>or<br>/var/tmp/profiledata/<profile> | Classical filesystem holding temporary and cache files, program preferences, “Dot files” that can be reconstructed (.rnd, .kde, .cddb) etc |
| /home/<profile>/noprotection | → /profiledata/<profile>/noprotection<br>Files the user decides to exclude from protection |

With Windows[37], something similar:

| | |
|---|---|
| /Users/<profile> | DPRFS (“On D:”) |
| /Profiledata/<profile> | NTFS (“On C:”) |
| /Users/<profile>/noprotection | → /Profiledata/<profile>/noprotection |

Another option would be to keeping either /dprfs or /profiledata within the other as with /home/<profile>/profiledata/ or /home/<profile>/dprfs/. However this would lead to inappropriate decisions for non-dprfs aware software. By example: in the former, /home/<profile>/.cache would see .cache unnecessarily protected by dprfs. In the latter, /home/<profile>/.ssh would end up unprotected from data-poisoning.

Where it's the behaviour of a program to create a default dot file that could be regenerated automatically but which might expect to be modified manually – such as ~/.bashrc – it would be acceptable to store the first copy on the dprfs volume or softlink it to then non-dprfs volume.

## Beyond AT Attachment

Hard drives provide long-term storage, and RAM short-term storage.

With the advent of flash memory, particularly in USB drives and SSDs, a unification of how the two types of storage interact with the main computer is possible.

[36] Filesystem Hierarchy Standard homepage http://www.linuxfoundation.org/collaborate/workgroups/lsb/fhs
[37] Windows observes a different standard that doesn't appear to have a name. Best guess: “The Windows XP Layout” http://www.quepublishing.com/articles/article.aspx?p=101731

This could see what passes to the drive expand to look more like filesystem operations – open(), pwrite(), pclose(), unlink() -- on top of the ATA commands. [not sure!]

## Devices

The highest form of dprfs would come from a long term storage device that honours dprfs: for instance, an SSD that would flag deletes instead. It would disable destructive behaviour (deletes, firmware changes) without evidence of physical presence, such as the firmware requiring a code be passed to it that's stamped on its device label. As obtaining that code requires examining the label, proof of presence is established. Such a device could therefore offer the highest data-safety even to a completely compromised host. Although, see http://www.theregister.co.uk/2015/02/17/kaspersky_labs_equation_group/

The next best form would be from remotely hosting a normal disk using dprfs. Such network storage is already available via routers [http://www.cnet.com/news/top-five-wi-fi-routers-with-built-in-network-storage/} and NAS drives [http://www.pcworld.com/article/2046116/qnap-ts-469-pro-review-a-fast-nas-box-well-suited-to-both-business-and-pleasure.html]; perhaps less usefully it could be offered by printers and AV centres. {Printer with secondary USB: http://www.product-reviews.net/2014/11/21/hp-officejet-pro-6835-review-with-aio-printer-specs/; AV center with bidirectional USB www.amazon.co.uk/dp/B00VX9GY8O]

The least secure form would be to host an ordinary hard drive using a dprfs driver to offer dprfs facilities. While better than nothing, compromising the hosts' direct device driver, or its dprfs driver would undo the valuable work done – to the point where this facility shouldn't be described as dprfs compliant.

## Shell changes

Right click | beyond use

Revert animation

## *Filesystem interface changes*

### int set_beyond_use(char *pall, timestamp)

Calculate and store a timestamp in the linkedlist metadata indicating the linkedlist is to be treated as "beyond use" at a specific point.

Timestamp may be relative or absolute. If relative, the host should calculate the absolute date before storing it.

On success return 0, otherwise return -1

### int get_payload_hash(char *buf, int buflen, char *pall)

Retrieve the hash of the payload associated with the path and linkedlist referenced by pall, and store it in the buffer provided at buf. If the hash is longer than buflen – 1, then store only the first buflen – 1 characters. Append \0.

This call helps determine whether a payload that could be uploaded is actually different from the existing payload without risking duplicating an existing payload, nor conducting a full read of that payload.

The hash itself is calculated by the host from the payload uploaded – it cannot be set independently of changing that payload.

On success return 0, otherwise return -1

## truncate/ftruncate

These should support negative lengths whereby a length of “-1000” causes all but the last 1000 bytes to be discarded, or the file left-padded with \0 to create a length of 1000 bytes.

This provides to programs a similar facility to tail(1)[38]

This would be of particular assistance when dealing with log files.

[38] “tail - output the last part of files” http://linux.die.net/man/1/tail